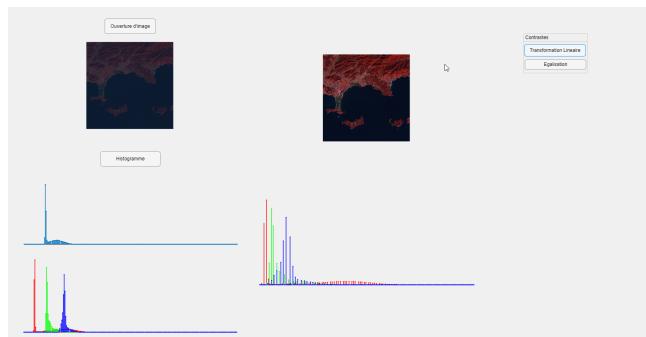

Instrumentation, capteurs, vision

TP : IHM et traitement d'image



Léa Yeromonahos

Table des matières

Introduction	2
Traitement d'image avec MatLab	3
Création de l'interface homme-machine (IHM)	3
IHM et traitement d'image	3
Ouverture d'une image	3
Transformations	9
Transformation linéaire	9
Égalisation	10
Contours	12
Filtrage de Prewitt : horizontal, vertical et mixte	13
Filtrage de Prewitt : horizontal, vertical et mixte	14
Comparaison des méthodes	16
Filtrage de bruit	16
Différents types de bruit	16
Différents types de filtrages	18
Comparaison entre les bruits et les filtrages	21
Seuillage	22
Binarisation	22
Seuillage multiseuils en niveau de gris	25
Seuillage multiseuils en couleur	26
Morphologie mathématique	27
Application au filtrage de bruit	30
Application à la détection de contours	31
Test de l'IHM sur d'autres images couleurs	32
Webcam	33
Traitement d'image en Python	35
Conclusion	41

Introduction

Le but de ce projet est de créer une interface homme machine en Matlab, avec l'outil de création **App Designer**. Cette IHM permettra à l'utilisateur d'appliquer différents traitements à une image.

Pour la partie Python, on utilisera différentes librairies nous permettant un travail plus industriel : le comptage d'objets.

Traitement d'image avec MatLab

Création de l'interface homme-machine (IHM)

Pour créer une IHM sur Matlab, on utilise l'**app designer** (anciennement GUIDE) :



FIGURE 1 – Lancement de l'app designer

Une fois l'app designer sélectionné, on arrive sur la page du menu, où l'on sélectionne la **blank app**, soit sans modèle :

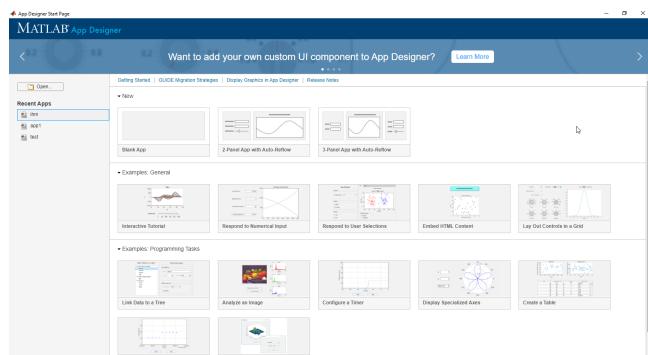


FIGURE 2 – Menu de l'app designer

Enfin, on arrive sur la page de création de notre future IHM, qui pour le moment est vide :

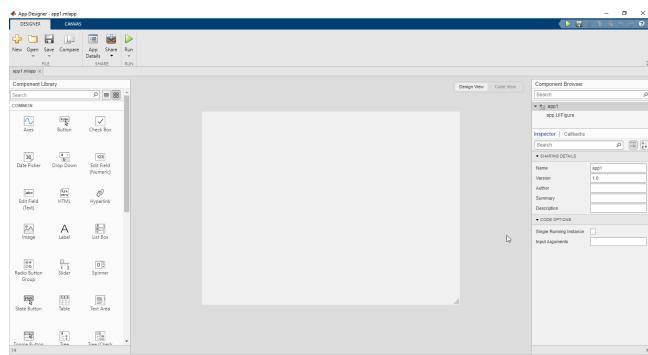


FIGURE 3 – Interface vide

IHM et traitement d'image

Ouverture d'une image

Afin de pouvoir ouvrir une image, on se place dans le **Design View** puis on commence par créer un bouton "Ouverture d'image" en plaçant un **push button** sur notre interface :

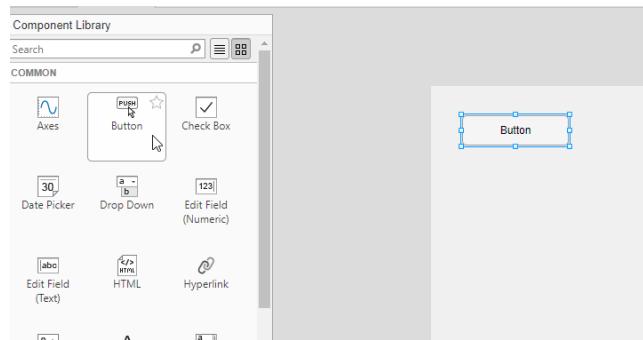


FIGURE 4 – Création du bouton sur l'interface

Une fois le bouton placé, on peut lui donner un nom dans la fenêtre **Component Browser** à droite, puis dans la section "BUTTON" :

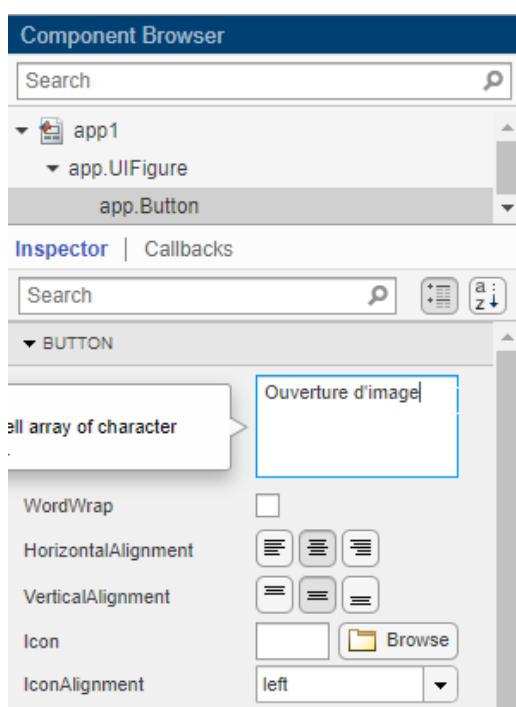


FIGURE 5 – Choix du nom du bouton

Une fois le nom donné, on assigne à ce bouton une fonction *callback*, c'est ce qui se passera lorsque l'on appuiera dessus :

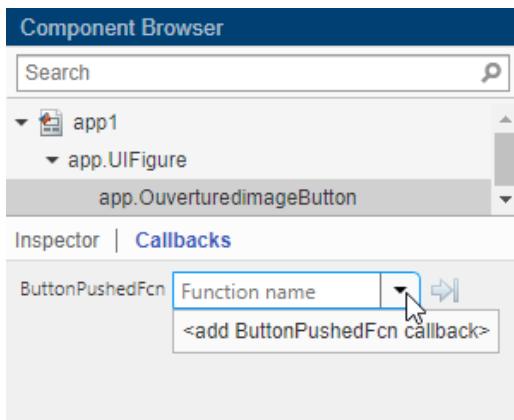


FIGURE 6 – Création d'une fonction callback

Lorsque la fonction est assignée, on arrive dans le **Code View** où l'on peut écrire notre fonction :

```

1 classdef app1 < matlab.apps.AppBase
2
3 % Properties that correspond to app components
4 properties (Access = public)
5     UIFigure             matlab.ui.Figure
6     OuvertureimageButton matlab.ui.control.Button
7 end
8
9 % Callbacks that handle component events
10 methods (Access = private)
11
12     % Button pushed function: OuvertureimageButton
13     function OuvertureimageButtonPushed(app, event)
14         %
15     end
16 end
17

```

FIGURE 7 – Fonction callback

La procédure sera la même pour chaque objet nécessitant une interaction (bouton, menu déroulant, liste, etc).

Afin que l'utilisateur puisse ouvrir une image, on doit d'abord accéder au fichier désiré, puis on récupère le nombre de lignes, colonnes et bandes dont on veut faire le traitement :

```

clc; close all
name_file=uigetfile( '*.img' )
Fid = fopen(name_file);
L = inputdlg('Lignes');
C = inputdlg('Colonnes');
B = inputdlg('Bandes');
L = str2double(L);
B = str2double(B);
C = str2double(C);
donnees = fread(Fid, C * L * B, '*ubit8');
fclose(Fid);
donnees = reshape(donnees, C, L, B);
donnees(:,:,1)=donnees(:,:,1)';
donnees(:,:,2)=donnees(:,:,2)';
donnees(:,:,3)=donnees(:,:,3)';
IM(:,:,1)=donnees(:,:,3);
IM(:,:,2)=donnees(:,:,2);
IM(:,:,3)=donnees(:,:,1);

```

Si on veut l'afficher, on crée d'abord dans notre interface un espace dédié à l'affichage :

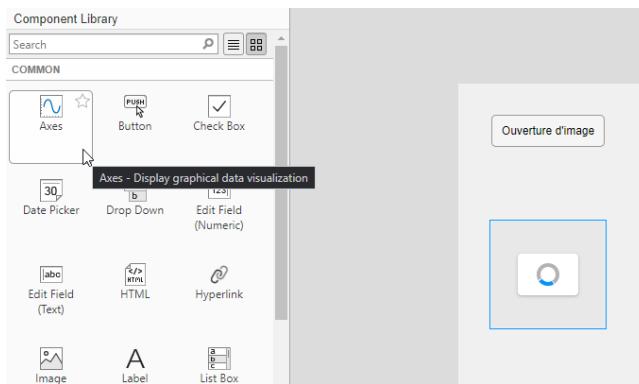


FIGURE 8 – Création de l'espace "Axe"

Puis on lui donne le nom qui nous permettra de l'appeler plus tard :

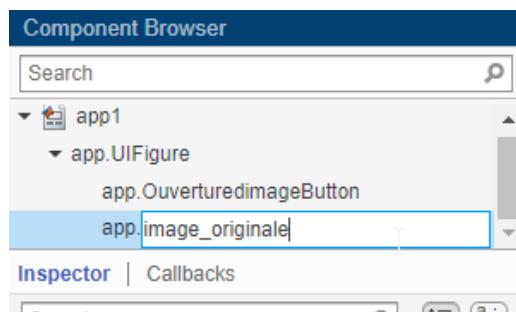


FIGURE 9 – Choix du nom de l'axe

On retourne ensuite dans le **Code View** et on ajoute les lignes suivantes :

```
imshow(IM, 'Parent', app.image_originale)
assignin("base", "IM", IM)
```

Avec la fonction *imshow()*, on affiche notre image **IM**, et l'on désigne l'espace créé précédemment, l'axe, comme *Parent*, c'est-à-dire l'endroit où sera envoyé l'image.
 Enfin, la fonction *assignin()* permet de créer une variable globale (une variable que l'on peut retrouver dans toutes nos fonctions) de notre image, qui sera stockée dans le workspace de base.

En lançant notre IHM, si on appuie sur le bouton "Ouverture d'image", on obtient, pour $1000 \times 1000 \times 3$ (1000 lignes, 1000 colonnes et 3 bandes) :

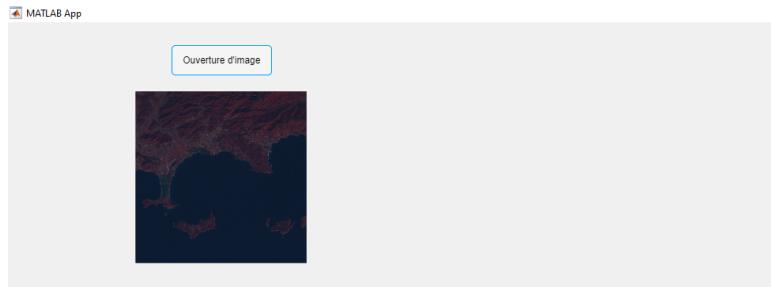


FIGURE 10 – Résultat de l'ouverture d'image

Maintenant que notre image est affichée, on veut pouvoir visualiser son histogramme. On crée pour cela une fonction histogramme dans notre application, en allant dans la partie **Code Browser** :

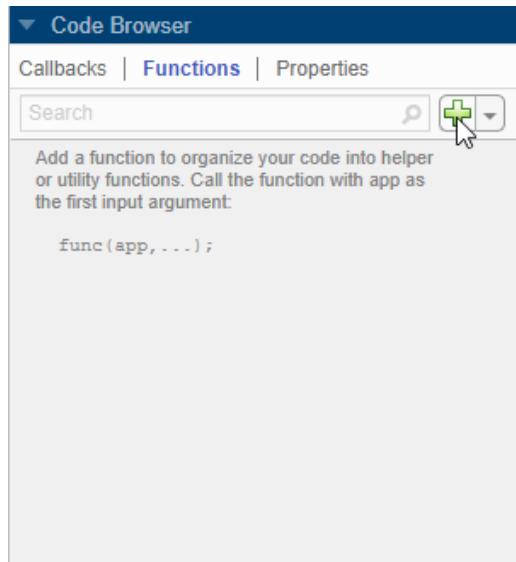


FIGURE 11 – Panel des fonctions de l'IHM

Lorsque l'on crée une nouvelle fonction, celle-ci s'ajoute dans le code de l'IHM tel que :

```
methods (Access = private)

function results = func(app)
end
end
```

FIGURE 12 – Création de la nouvelle fonction

On y crée donc la fonction *histogramme()*, qui prend en paramètre une image. Comme nous sommes dans une application MatLab, on ajoute en premier argument *~*, qui nous permet d'utiliser cette fonction dans tout l'IHM :

```
function h = histogramme(~, img)
h = zeros(256, 1);
[nl, nc] = size(img);
for i = 1:nl
    for j = 1:nc
        val = img(i, j) + 1;
        h(val) = h(val) + 1;
    end
end
end
```

On commence par créer un vecteur colonne de 0, de taille 256 (pour des pixels allant de 0 à 255), puis on récupère la taille de l'image avec la fonction *size()*.

On récupère ensuite la valeur de chacun des pixels, que l'on stocke dans notre vecteur **h**, qui sera notre histogramme. On ajoute +1 pour ne pas décaler les valeurs car on commence notre histogramme à 1

(les boucles dans MatLab sont définies ainsi).

Notre histogramme ne peut être calculé que pour une bande, donc soit l'image en argument est en nuances de gris, soit on applique cette fonction bande par bande pour avoir l'histogramme couleur.

Maintenant, pour afficher un histogramme, il nous faut créer un espace dédié à l'affichage de graphiques. On choisit donc **Axes** dans **Design View**, puis on le place sur notre IHM. On lui donne un nom qui nous permettra de l'appeler par la suite.

Une fois cela fait, on se place dans le **callback** de notre bouton "Ouverture d'image". Pour un histogramme de l'image en nuances de gris, on la transforme d'abord en N/B avec la fonction *im2gray()*, puis on y applique la fonction *histogramme()* :

```
imgNB = im2gray(IM);
hNB = histogramme(app, imgNB);
stem(app.hist_avant, hNB, '.')
```

En argument de notre fonction *histogramme()*, il y a évidemment notre image, et le mot-clef **app** qui était symbolisé par `~` lors de sa déclaration.

Pour afficher l'histogramme, on utilise ici la fonction *stem()* qui affiche point par point, et on y déclare en premier argument l'endroit où l'on veut l'afficher, soit notre **Axes** de tout à l'heure.

On obtient finalement :

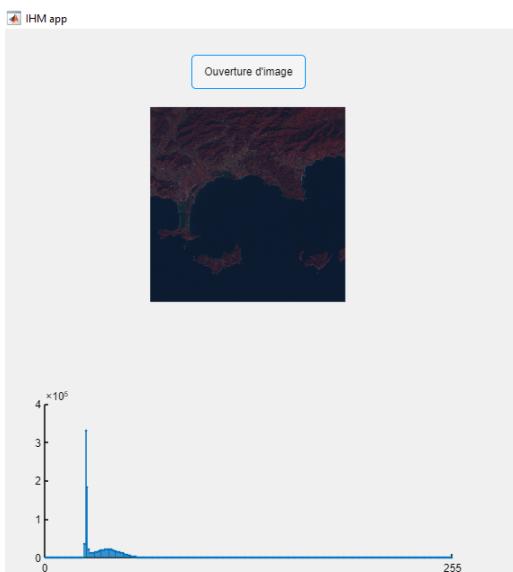


FIGURE 13 – Histogramme de l'image originale en N/B

L'histogramme ne comporte qu'un mode, concentré dans les valeurs basses. Cela correspond avec notre image originale qui est très sombre.

Maintenant, si on veut l'histogramme en couleur, on réalise la même chose mais pour chaque bande de l'image :

```
hR = histogramme(app, IM(:, :, 1));
hG = histogramme(app, IM(:, :, 2));
hB = histogramme(app, IM(:, :, 3));

stem(app.hist_avant_rgb, hR, '.', 'r')
hold(app.hist_avant_rgb, 'on')
stem(app.hist_avant_rgb, hG, '.', 'g')
stem(app.hist_avant_rgb, hB, '.', 'b')
hold(app.hist_avant_rgb, 'off')
```

On prend donc chaque bande de l'image originale en couleur, puis on représente chaque histogramme

(un pour chaque couleur RGB). La fonction `hold()` nous permet d'afficher plusieurs éléments sur un même graphique.

On obtient le résultat final :

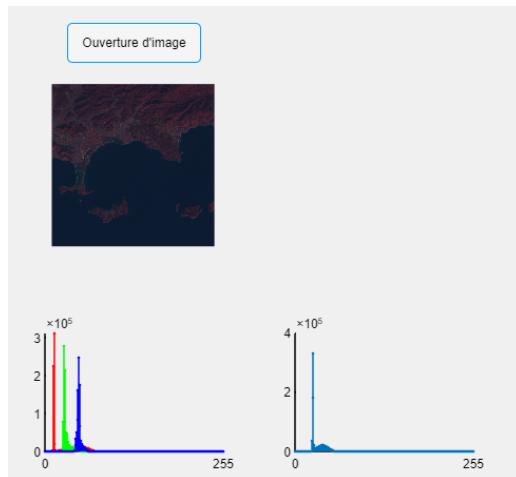


FIGURE 14 – Histogrammes couleur et N/B de l'image originale

L'image étant sombre, on observe que les couleurs R, G et B sont concentrées dans les valeurs basses.

Transformations

On crée un panel "Transformations" qui comportera la Transformation linéaire et l'égalisation de l'histogramme. Pour cela, on sélectionne le **panel** dans le **Design View** :

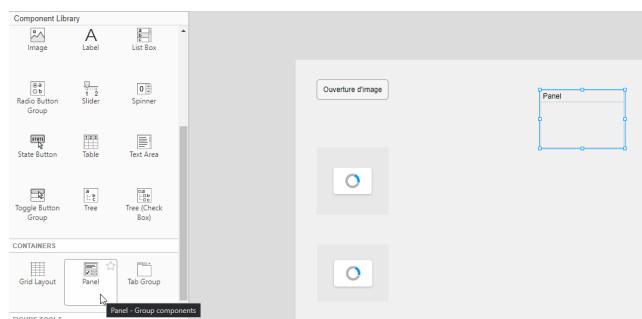


FIGURE 15 – Création d'un panel

On y ajoute les boutons pour les deux transformations et on peut ensuite écrire leur fonction.

Transformation linéaire

On récupère d'abord l'image originale avec `evalin()`, qui fonctionne de pair avec `assignin()`. On crée ensuite une fonction à part (comme pour `histogramme()`) qui va réaliser la transformation linéaire :

```
function IM2 = transf_lineair(~, img)
    ma = max(max(max(img)));
    ma = double(ma);
    mi = min(min(min(img)));
    mi = double(mi);
```

```

alpha = double((255/(ma-mi)));
alpha = double(alpha);
beta = double(((255*mi)/(ma-mi)));
beta = double(beta);
IM2 = round(alpha.*img + beta);
end
    
```

On récupère le maximum et le minimum des tous les pixels de l'image, puis on crée deux variables α et β , telles que :

$$\alpha = \frac{255}{\max - \min} \quad \beta = \frac{-255 \times \min}{\max - \min}$$

Enfin, la nouvelle image aura, pour chaque pixel, une nouvelle valeur donnée par l'équation :

$$\text{Nouvelle image} = \alpha \times \text{Ancienne image} + \beta \quad (1)$$

Comme α et β ne sont pas des entiers, on utilise la fonction *round()* qui prend l'arrondi supérieur pour avoir une image composée de valeurs entières. La nouvelle valeur minimale de l'image sera 0, et maximale 255.

On applique ce traitement à chaque bande de notre image, et on en calcule le nouvel histogramme :

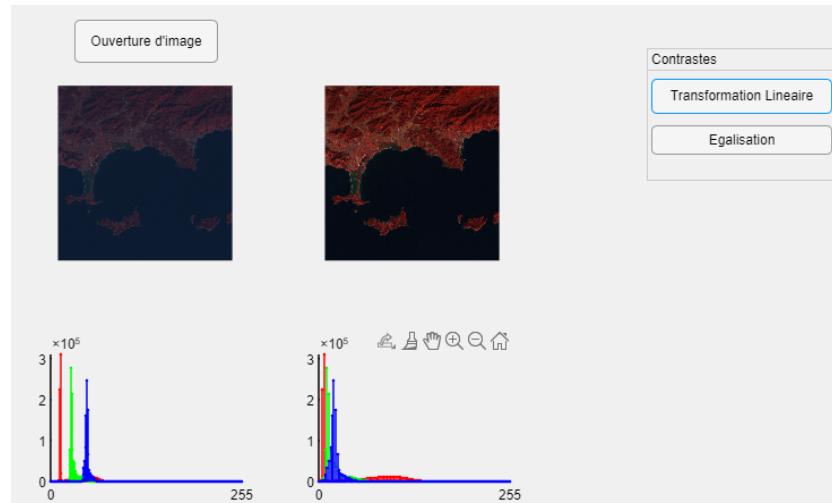


FIGURE 16 – Résultat de la transformation linéaire, nouvelle image et histogramme associé

L'image est plus claire, on peut y distinguer les détails. Les pixels rouges sont plus nombreux vers les valeurs centrales de l'histogramme, le vert et le bleu sont bien plus étalés. On utilisera dorénavant cette image comme entrée des futurs traitements.

Égalisation

Le principe est de calculer la probabilité P de chaque pixel différent de 0. On prend donc chaque valeur de l'histogramme que l'on divise par la somme des occurrences des pixels dans l'histogramme. La somme de P vaudra 1.

Ensuite, on calcule la somme S telle que :

$$S(i) = S(i-1) + P(i)$$

On part de $S(1)$ égal à la première valeur de P , puis $S(2)$ égal à $S(1)$ plus la valeur de $P(2)$, etc... Ces valeurs de S seront ensuite multipliées à 255 pour obtenir les nouvelles valeurs de l'image :

```

function IM2 = egalisation(~, img, h)
P = h./sum(h);
S = zeros(length(P), 1);
S(1) = P(1);
ma = max(max(max(img)));
ma = double(ma);

for i = 2:ma + 1
    S(i) = P(i) + S(i - 1);
end
for i = 2:ma
    if P(i - 1) == 0
        S(i - 1) = 0;
    end
end
CN = round(S.*255);

[nl, nc] = size(img);
IM2 = zeros(nl, nc);

for i = 1:nl
    for j = 1:nc
        IM2(i, j) = CN(img(i, j) + 1);
    end
end
IM2 = uint8(IM2);
end

```

En récupérant l'image précédente et son histogramme, on calcule les probabilités P , qui seront stockées sous forme d'un vecteur colonne.

On crée ensuite un vecteur colonne S de la taille de P , que l'on remplit avec une première boucle allant de 2 (car on va calculer $S(i-1)$) à la valeur maximale des pixels de l'image (**ma**) + 1. On initialise auparavant S avec la première valeur de P .

Ensuite, pour ne pas prendre en compte les 0 et ainsi décaler les valeurs de S par rapport à P , on crée une autre boucle allant de 2 à **ma** où tous les pixels de P de valeur 0 seront à 0 pour S .

Enfin, les nouveaux pixels de l'image seront les **CN**, soit les S multipliés par 255. On utilise la fonction *round()* pour obtenir des valeurs entières.

Maintenant, pour avoir notre image égalisée, on récupère les dimensions de l'image originales, puis on crée la nouvelle image vide avec ces dimensions.

Dans une double boucle, on rempli la nouvelle image **IM2** par les valeurs de **CN** à l'indice de l'image originale.

Enfin, on force le format *uint8*, qui est celui de base des images sur MatLab.

La fonction *egalisation()* est créée, on l'assigne à un bouton **Egalisation**, puis dans le callback :

```

img = evalin("base", "IM2");
hR = evalin("base", "hR");
hG = evalin("base", "hG");
hB = evalin("base", "hB");

IM2(:, :, 1) = egalisation(app, img(:, :, 1), hR);
IM2(:, :, 2) = egalisation(app, img(:, :, 2), hG);
IM2(:, :, 3) = egalisation(app, img(:, :, 3), hB);

imshow(IM2, 'Parent', app.figure_apres)

nhR = histogramme(app, IM2(:, :, 1));
nhG = histogramme(app, IM2(:, :, 2));
nhB = histogramme(app, IM2(:, :, 3));

```

```

stem(app.hist_apres, nhR, '.', 'r')
hold(app.hist_apres, 'on')
stem(app.hist_apres, nhG, '.', 'g')
stem(app.hist_apres, nhB, '.', 'b')
hold(app.hist_apres, 'off')
    
```

On récupère notre image qui a déjà subi la transformation linéaire, ainsi que l'histogramme de chacun de ses bandes. On y applique l'égalisation, puis on l'affiche, avec le nouvel histogramme correspondant.

On obtient :

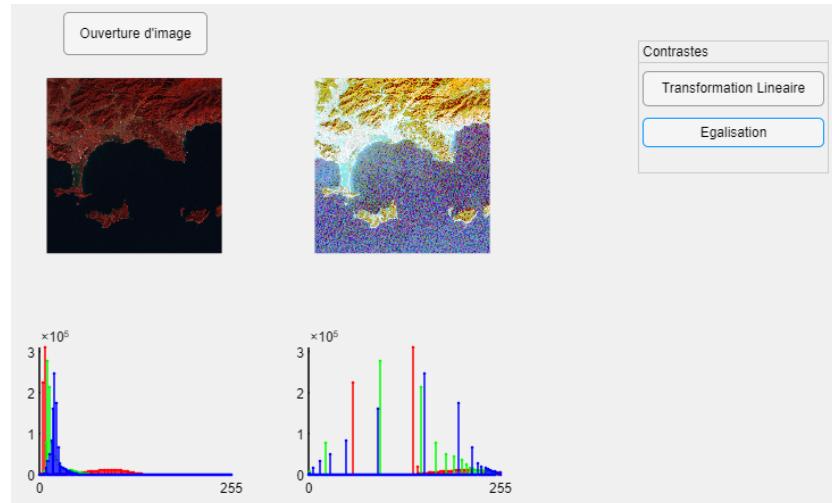


FIGURE 17 – Résultat de l'égalisation après transformation linéaire, nouvelle image et histogramme associé

L'image est beaucoup plus claire, mais paraît comme saturée. Lorsque l'on regarde l'histogramme, toute la dynamique est utilisée, et ce pour les 3 canaux R, G et B.

Contours

Afin de détecter les contours dans une image, on peut utiliser la méthode de filtrage de Prewitt/Sobel : on applique un masque (un filtre) par convolution à l'image. La convolution va changer la valeur du pixel central du masque sur l'image, selon que l'on choisisse un filtrage horizontal, vertical, ou mixte. Les matrices pour horizontal et vertical sont respectivement les suivantes :

$$Hs2 = \begin{bmatrix} 1 & c & 1 \\ 0 & 0 & 0 \\ -1 & -c & -1 \end{bmatrix} \quad Hs1 = \begin{bmatrix} 1 & 0 & -1 \\ c & 0 & -c \\ 1 & 0 & -1 \end{bmatrix}$$

La différence entre Prewitt et Sobel réside dans la valeur de c , valant respectivement 1 et 2. Un filtrage mixte est le résultat de la somme des deux, soit :

$$Hmixte = |Hs1| + |Hs2|$$

On va donc créer un nouveau panel "Contours" avec deux boutons, **Prewitt** et **Sobel**.

Filtrage de Prewitt : horizontal, vertical et mixte

Pour le filtrage de Prewitt, dans le callback associé, on fixe la valeur de c à 1, et on applique la convolution du masque **Hs1** pour le filtrage vertical, **Hs2** pour horizontal et enfin la somme des deux pour le mixte :

```

value = app.Prewitt.Value;
im = evalin("base", "IM");
im = rgb2gray(im);
c = 1;
Hs2 = [1 c 1; 0 0 0; -1 -c -1];
Hs1 = [1 0 -1; c 0 -c; 1 0 -1];
IMvert = conv2(im, Hs1);
IMvert = uint8(IMvert);
IMhor = conv2(im, Hs2);
IMhor = uint8(IMhor);
IMmix = abs(IMvert) + abs(IMhor);
IMmix = uint8(IMmix);
if value == "Prewitt"
    imshow(im, 'Parent', app.figure_apres)
elseif value == "Vertical"
    imshow(IMvert, 'Parent', app.figure_apres)
elseif value == "Horizontal"
    imshow(IMhor, 'Parent', app.figure_apres)
elseif value == "Mixte"
    imshow(IMmix, 'Parent', app.figure_apres)
end

```

La détection de contour s'effectue en noir et blanc, on récupère donc notre image ayant déjà subie la transformation linéaire, puis on utilise la fonction *im2gray()* pour la mettre en N/B.

Ensuite, on utilise la fonction *conv2()* qui va réaliser une convolution en 2 dimensions, entre l'image et le masque choisi.

Enfin, pour chaque choix sélectionné, on renvoie le filtrage correspondant.

On obtient, pour le vertical :

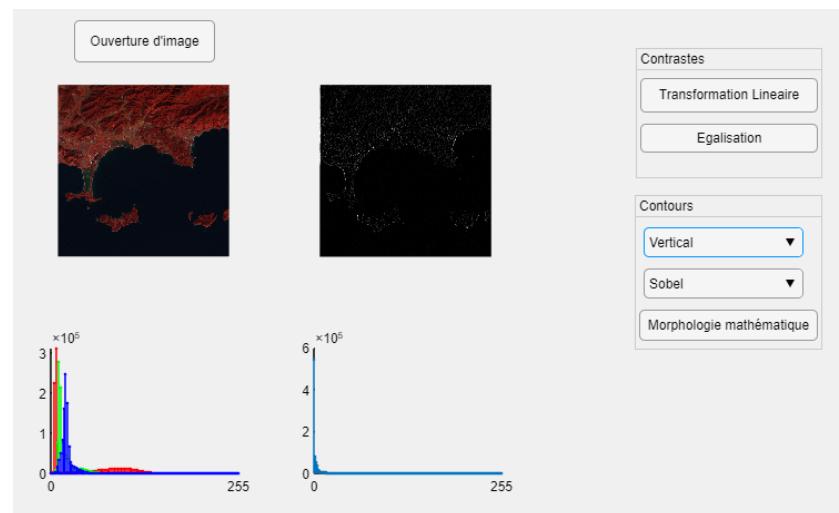


FIGURE 18 – Résultat filtrage de Prewitt vertical

Pour l'horizontal :

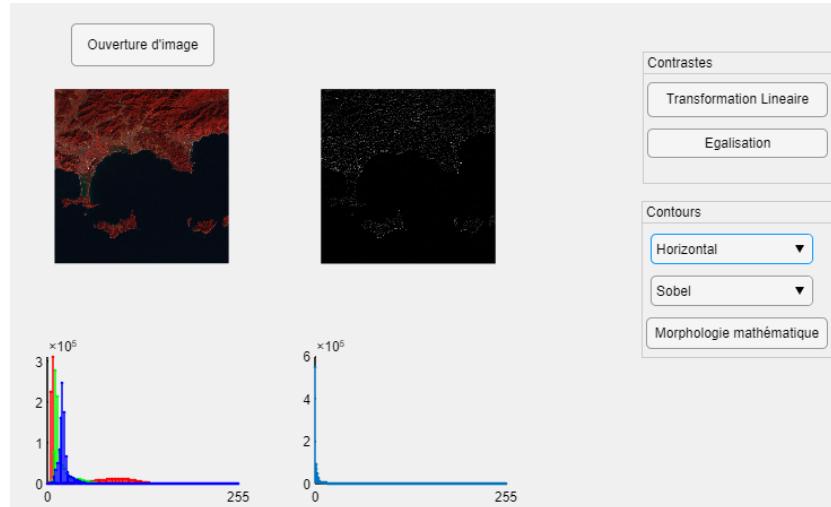


FIGURE 19 – Résultat filtreage de Prewitt horizontal

Et enfin pour le mixte :

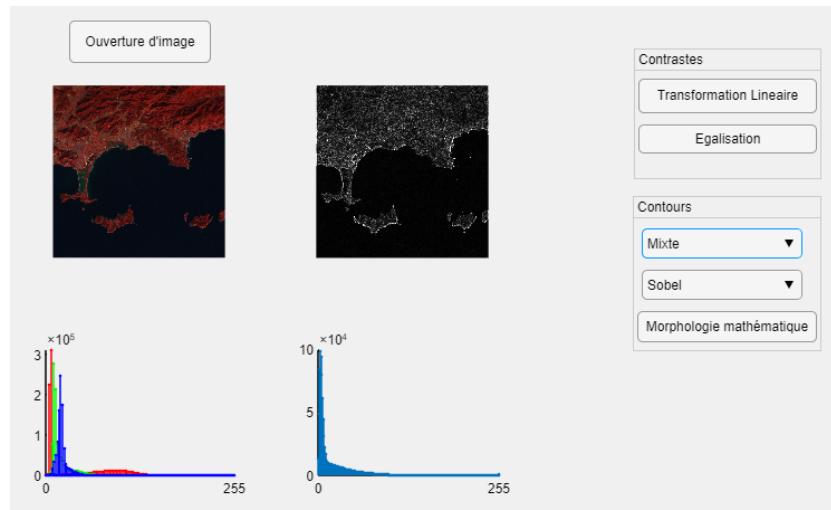


FIGURE 20 – Résultat filtreage de Prewitt mixte

Le filtrage mixte nous permet de réellement distinguer les contours, contrairement aux vertical et horizontal qui ne donnent pas assez de détails individuellement. L'horizontal nous permet d'observer les contours horizontaux principalement, et le vertical plutôt ce qui est vertical, comme le contour des îles par exemple.

Filtrage de Sobel : horizontal, vertical et mixte

Pour le filtrage de Sobel, la valeur de c est à 2. On obtient, pour le filtrage vertical :

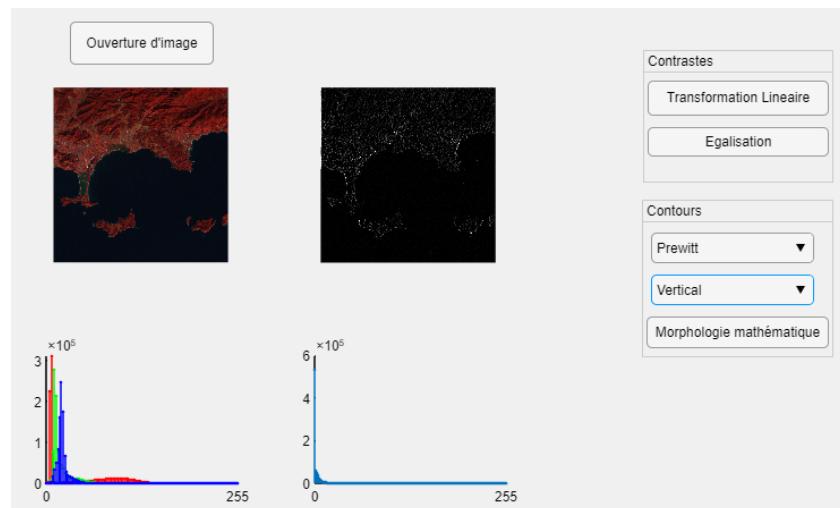


FIGURE 21 – Résultat filtreage de Sobel vertical

Pour le filtreage horizontal :

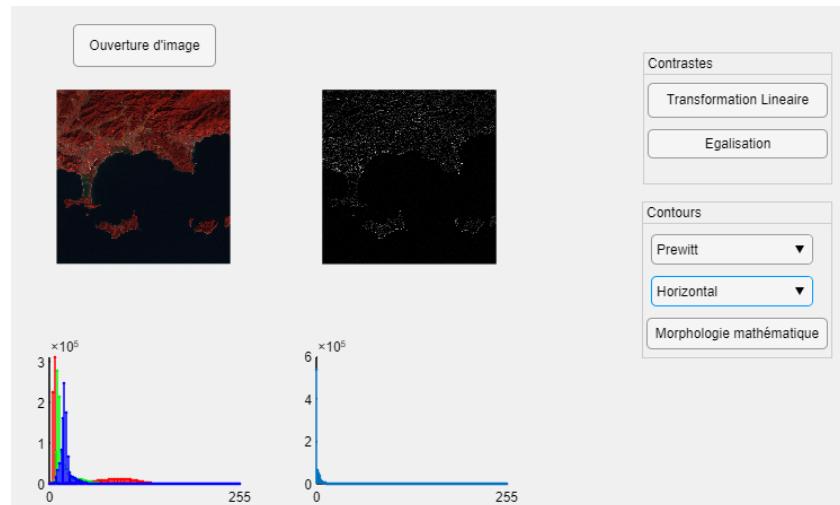


FIGURE 22 – Résultat filtreage de Sobel horizontal

Et enfin le mixte :

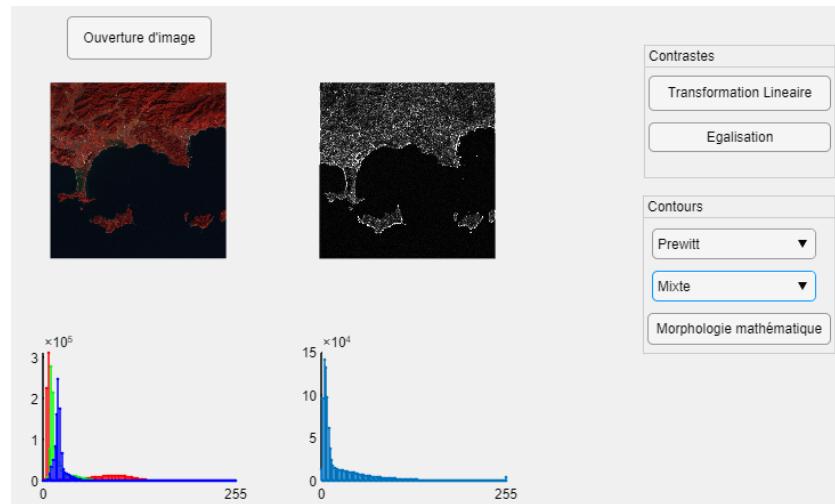


FIGURE 23 – Résultat filtreage de Sobel mixte

Avec le filtreage horizontal, on distingue plutôt les contours dans le continent, horizontaux, alors que le vertical nous permet de distinguer les contours verticaux des îles. Enfin, la somme des deux réalise une excellente détection des contours. Néanmoins, on peut observer quelques points blancs dans l'eau.

Comparaison des méthodes

La méthode de Sobel est plus efficace dans notre cas, avec un c à 2. Les contours sont très distincts, donnant une impression de "ville illuminée" plus forte qu'avec Prewitt. De plus, les filtrages vertical et horizontal de ce dernier ne nous permettent que très peu de deviner les contours, contrairement à ceux de Sobel.

Filtreage de bruit

Dans cette partie, on va d'abord créer quelques bruits classiques afin de rendre notre image bruitée, pour la filtrer par la suite.

On crée donc un panel "Filtreage" et l'on y ajoute, comme pour les contours, des boutons de listes déroulantes pour le **bruit** :

- gaussien
- poisson
- poivre et sel

et le **filtrage** :

- passe-bas
- médian
- milieu

Différents types de bruit

Le bruit gaussien se caractérise par des petits points sur l'image. On peut utiliser la fonction *imnoise()* avec comme paramètre "gaussien" afin d'en ajouter sur notre image, comme ceci :

```
IM = imnoise(im, "gaussian");
```

Et on obtient ainsi l'image :

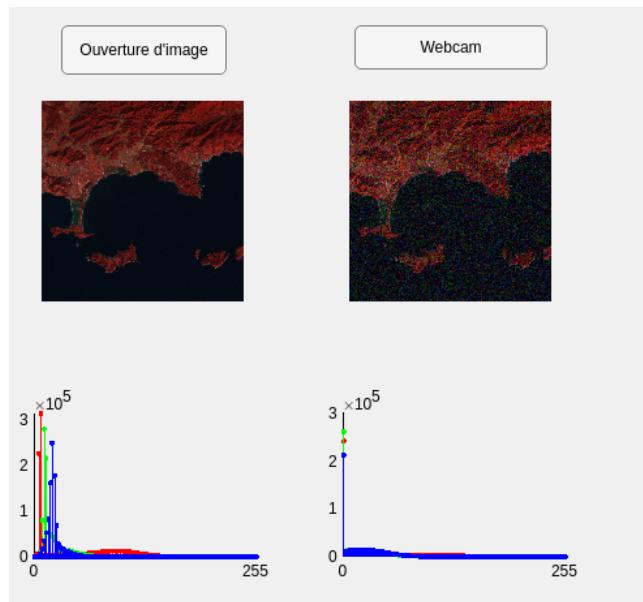


FIGURE 24 – Ajout de bruit gaussien sur l'image

L'image est visuellement moins nette.

Le bruit de poisson se caractérise lui par une image floue. De la même manière, on utilise la fonction `imnoise("poisson")` et on obtient :

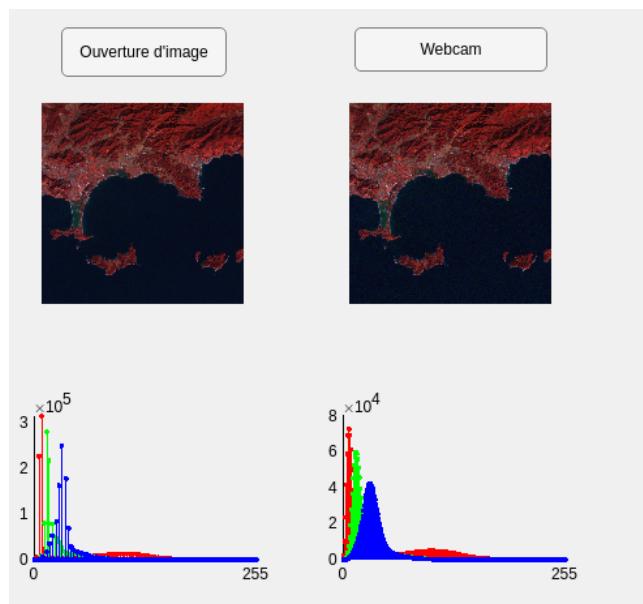


FIGURE 25 – Ajout de bruit de poisson sur l'image

Les modes de l'histogramme sont plus étalés, justifiant la notion de flou.

Enfin, le bruit poivre et sel est reconnaissable à ces tâches parsemant l'image, tel que :

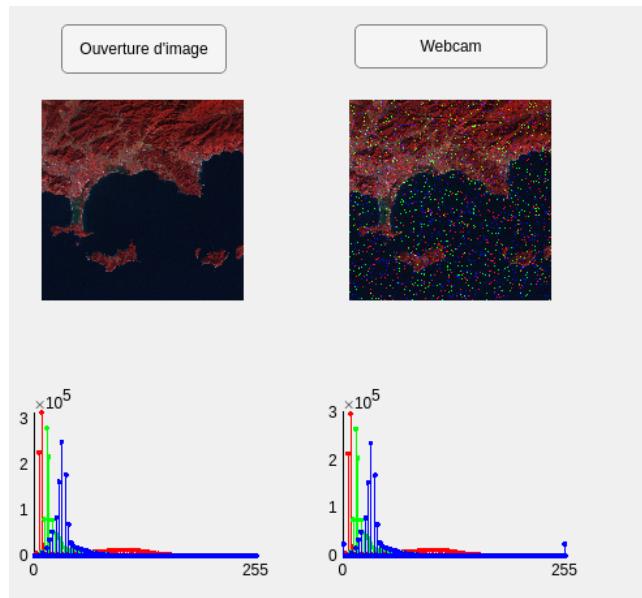


FIGURE 26 – Ajout de bruit poivre et sel sur l'image

Différents types de filtrages

Le filtrage passe-bas consiste à appliquer un masque moyenneur sur l'image bruitée, tel que :

$$\text{Masque} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \times \frac{1}{9} \quad (2)$$

pour un masque de taille $T = 3$.

On réalise donc une convolution entre l'image et le masque, soit :

```
T = 3;
fil = (1/9)*ones(T, T);

IM(:,:,1) = conv2(img(:,:,1), fil);
IM(:,:,2) = conv2(img(:,:,2), fil);
IM(:,:,3) = conv2(img(:,:,3), fil);
```

On utilise la fonction *conv2()* qui réalise une convolution en 2-D (nécessaire pour une image), avec comme arguments l'image originale et le masque **fil**. On obtient, pour une image bruitée par un bruit gaussien :

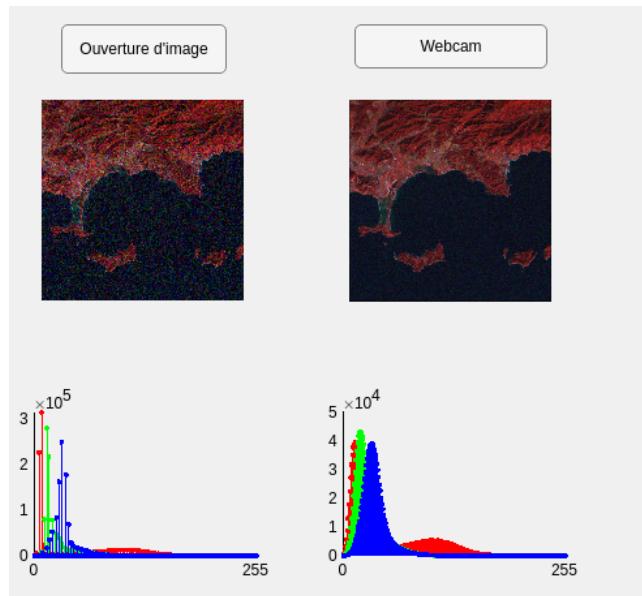


FIGURE 27 – Filtrage passe-bas sur un bruit gaussien

Le filtre passe-bas a tendance à étaler les zones, comme on peut le voir sur l'histogramme. On peut en déduire qu'il aura tendance à rendre les contours assez flous.

Le filtre milieu consiste à prendre la valeur moyenne des pixels dans une zone de l'image délimitée par un masque de taille T :

```
function IM = filtre_milieu(~, img, T)
    [nl, nc] = size(img);
    IM = img;
    for i = 2:nl-1
        for j = 2:nc-1
            v = img(i-1:i+1, j-1:j+1);
            v = reshape(v, 1, T^2);
            v = sort(v);
            IM(i, j) = (v(1)+v(9))/2;
        end
    end
end
```

On récupère les pixels selon la taille du masque, on les place tous dans un vecteur avec la fonction `reshape()`, on trie ce vecteur avec `sort()` et on récupère la valeur moyenne.

On obtient le résultat suivant, pour un bruit gaussien :

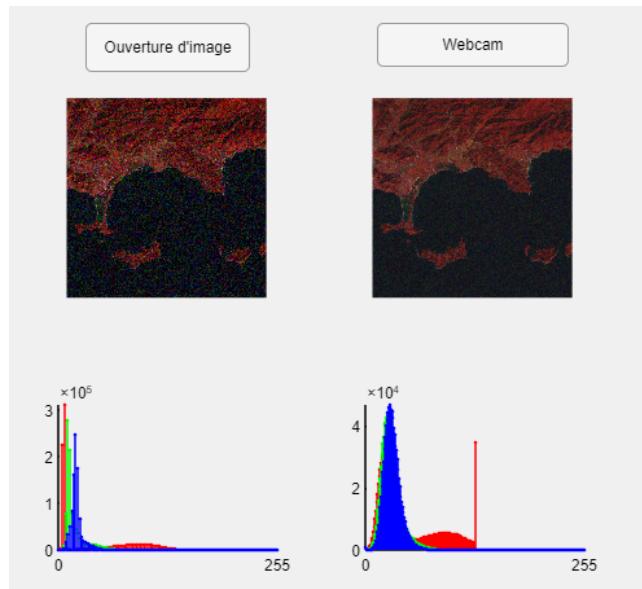


FIGURE 28 – Filtrage milieu sur un bruit gaussien

Le filtre milieu ne semble pas adapté à ce type de bruit. Sinon, on remarque qu'il a tendance à étaler les contours, rendant les détails de l'image plus flous.

Enfin le filtre médian repose sur le même principe que le milieu, si ce n'est qu'au lieu de prendre la valeur moyenne, on prend la médiane :

```
function IM = filtre_median(~, img, T)
    [nl, nc] = size(img);
    IM = img;
    for i = 2:nl-1
        for j = 2:nc-1
            v = img(i-1:i+1, j-1:j+1);
            v = reshape(v, 1, T^2);
            v = sort(v);
            IM(i, j) = v(5);
        end
    end
end
```

Et on obtient donc, toujours pour un bruit gaussien :

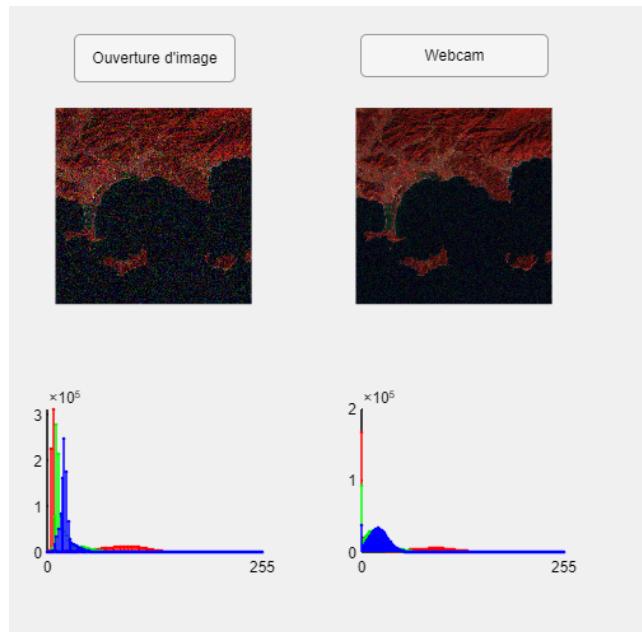


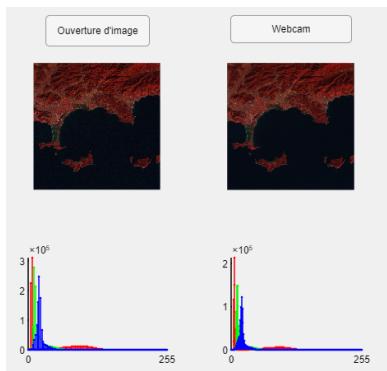
FIGURE 29 – Filtrage médian sur un bruit gaussien

L'histogramme pour le filtre médian indique qu'un bon nombre de pixels est à 0. On peut l'expliquer par le fait que comme l'image est assez sombre (la partie mer est bleu foncé), lors du calcul de la valeur médiane, on est assez souvent sur un vecteur de pixels sombres, augmentant ainsi la probabilité d'obtenir une valeur médiane basse.

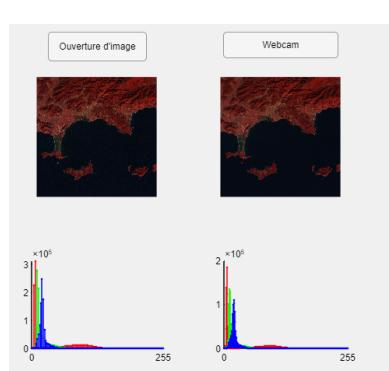
Comparaison entre les bruits et les filtrages

Pour un bruit de type gaussien, le plus efficace des filtrages proposés est le passe-bas. Les filtres milieu et médian laissent trop de bruit.

Pour un bruit de Poisson, le meilleur filtre semble être le passe-bas, bien qu'avec le médian il y ait peu de différences. Par contre, le milieu ne semble pas adapté car il "ajoute" des couleurs qui ne devraient pas être là à certains endroits :



(a) Filtre passe-bas sur un bruit de Poisson



(b) Filtre médian sur un bruit de Poisson

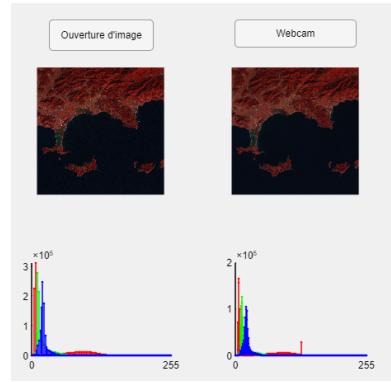
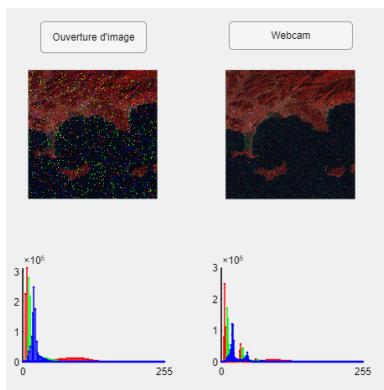
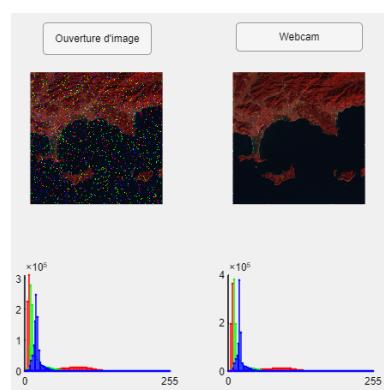


FIGURE 31 – (c) Filtre milieu sur un bruit de Poisson
 Comparaison des filtres sur un bruit de Poisson

Enfin, pour un bruit poivre et sel, le meilleur filtre est le médian. Il enlève tous les défauts de l'image, alors que le filtre milieu étale le bruit et le passe-bas laisse des pixels colorés aléatoires :



(a) Filtre passe-bas sur un bruit Poivre et Sel



(b) Filtre médian sur un bruit Poivre et Sel

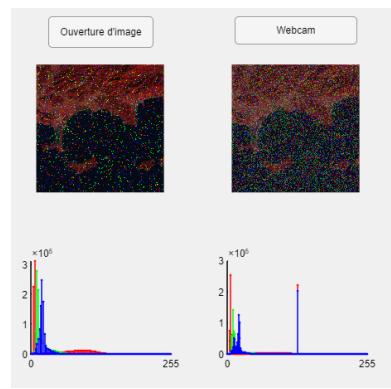


FIGURE 33 – (c) Filtre milieu sur un bruit Poivre et Sel
 Comparaison des filtres sur un bruit Poivre et Sel

Seuillage

Binarisation

L'objectif est de répartir les pixels de l'image en noir ou en blanc, selon leur valeur comparée à un seuil fixé. Par exemple, si pour un seuil à 40 le pixel traité vaut 39, alors il sera noir. Par contre, son voisin valant 78 sera blanc.

La valeur du seuil est laissée au choix de l'utilisateur, au moyen d'un **slider bar** (ou curseur défilant), compris entre 0 et 255.

Une fois le curseur placé sur l'ihm, on lui assigne une fonction callback, se référant à la **valeur changée**, soit *ValueChangedFcn*, ce qui nous permettra de récupérer à chaque déplacement du curseur la nouvelle valeur de seuil :

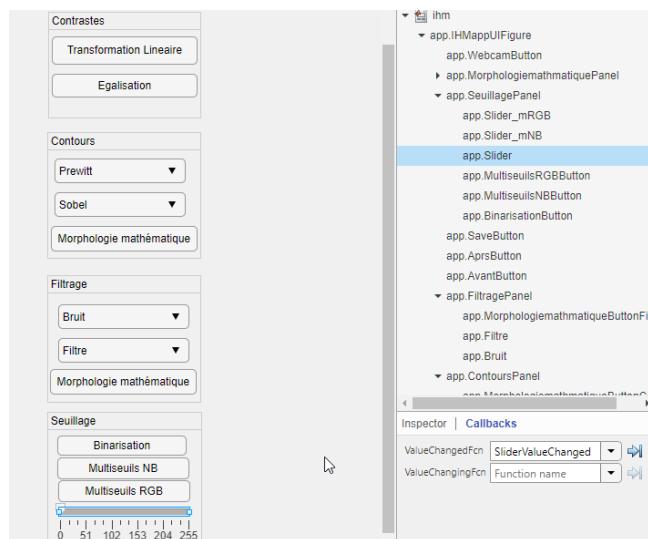


FIGURE 34 – Création du curseur défilant pour la binarisation

On aura donc deux fonctions callbacks pour le traitement par binarisation : une première correspondant au bouton **binarisation**, qui en réalité ne sert qu'à rendre accessible le slider, et la deuxième correspondant au **changement de valeur** du curseur. C'est dans la deuxième que les traitements seront faits.

Le code de la première fonction callback est court :

```
app.Slider.Visible = "on";
app.Slider.Enable = "on";
```

Le nom de notre curseur défilant est **Slider**. Cette fonction, comme dit plus haut, permet simplement de rendre accessible le curseur.

La deuxième fonction contient le traitement, et une fonction *binarisation()* telle que :

```
function IM = binarisation(~, img, val)
[nl, nc] = size(img);
for i = 1:nl
    for j = 1:nc
        if img(i, j) > val
            IM(i, j) = 1;
        else
            IM(i, j) = 0;
        end
    end
end
IM = IM.*255;
end
```

On peut maintenant réaliser la binarisation de l'image.

On récupère donc d'abord la valeur du seuil, puis on applique la fonction *binarisation()* à notre image, sur la bande 1 (PIR). On calcule aussi le nouvel histogramme, et on obtient, pour un seuil à 23 :

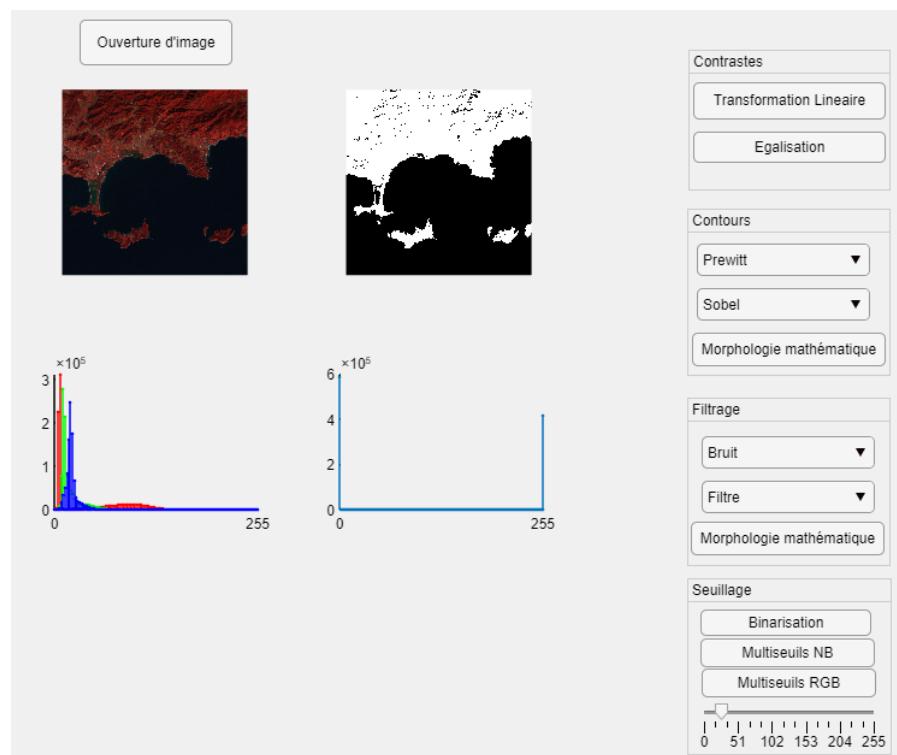


FIGURE 35 – Résultat de la binarisation, avec un seuil à 23

La terre et l'eau sont distinguement séparées, et l'histogramme affiche un grand nombre de pixel dans le blanc.

Avec maintenant un seuil à 102 :

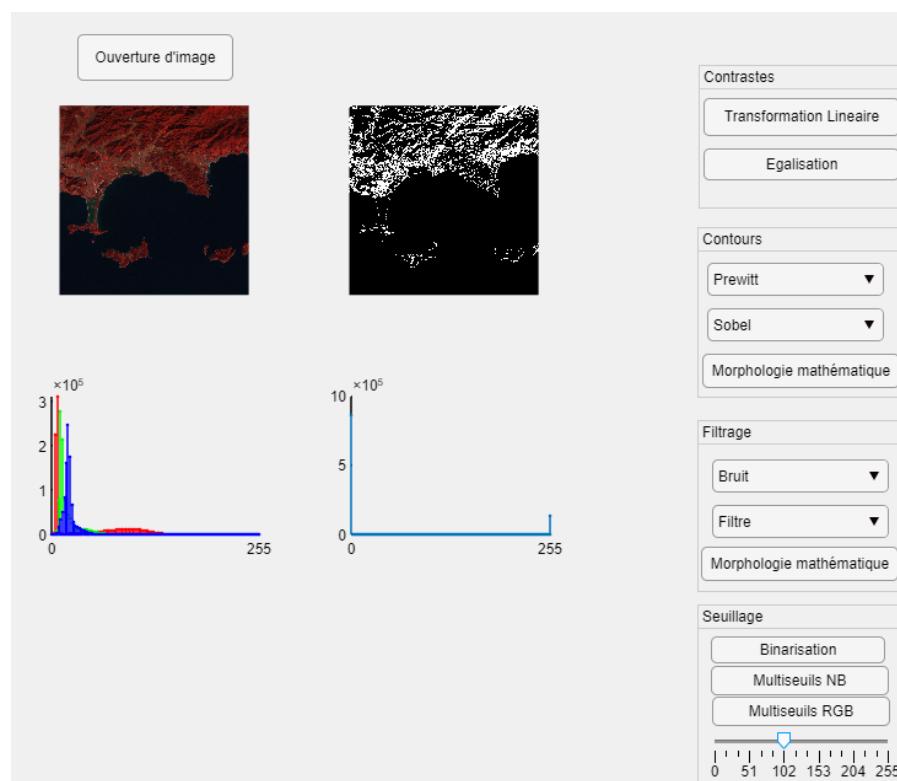


FIGURE 36 – Résultat de la binarisation, avec un seuil à 102

Cette fois, on perd un peu la liaison entre la terre et les îles, et une partie de la côte est tâchée de noir. On peut le vérifier avec l'histogramme qui affiche beaucoup moins de pixels blancs que noirs. Plus on augmente le seuil, et plus le nombre de pixels noirs augmente.

Seuillage multiseuils en niveau de gris

Le seuillage multiseuils consiste à répartir les pixels d'une image équitablement selon le niveau de seuil choisi. Par exemple, si on a un seuil à 3, alors on aura 85 pixels par seuil. Les pixels de l'image entre 0 et 85 vaudront 0, ceux entre 86 et 170 vaudront 128 et les derniers vaudront 255. L'objectif est de créer une fonction qui calcule ces nouvelles valeurs en fonction du seuil choisi par l'utilisateur. Du côté IHM, on reste sur une slider bar, de la même manière que pour la binarisation. Pour la fonction multiseuillage N/B :

```

function IM = multiseuillage_NB(~, img, seuil)
[nl, nc] = size(img);
seuil = round(seuil);

listPixel = zeros(nl*nc, 1);
for l = 1:nl
    for c = 1:nc
        listPixel(nc * (l - 1) + (c - 1) + 1) = img(l,c);
    end
end

listPixel = sort(listPixel, 1);

IM = zeros(nl, nc);
for l = 1:nl
    for c = 1:nc
        % Algo de recherche par dichotomie
        mini = 1;
        maxi = nc * nl;
        index = 1;
        founded = false;
        while ~founded && mini <= maxi
            middle = round((maxi + mini) / 2);
            if listPixel(middle) == img(l, c)
                founded = true;
                index = middle;
            elseif img(l, c) > listPixel(middle)
                mini = middle + 1;
            else
                maxi = middle - 1;
            end
        end

        index = floor((index * seuil) / (nl * nc));
        grayValue = floor((255 * index) / (seuil - 1));
        IM(l, c) = grayValue;
    end
end
IM = round(IM);
IM = uint8(IM);
end

```

On met le seuil en paramètre de la fonction, soit **seuil**. On récupère les dimensions de l'image à traiter, puis on stocke les pixels de l'image dans un vecteur que l'on trie.

Ensuite, on utilise un algorithme de recherche dichotomique afin de déterminer la valeur finale du pixel selon son ordre dans le vecteur trié. L'explication complète de l'algorithme se trouve [ici](#).

On répartit maintenant les pixels selon les seuils et leur valeur calculée plus haut, puis en fin de traitement on force les valeurs entières et le format uint8.

On obtient, pour un seuil à 4 :

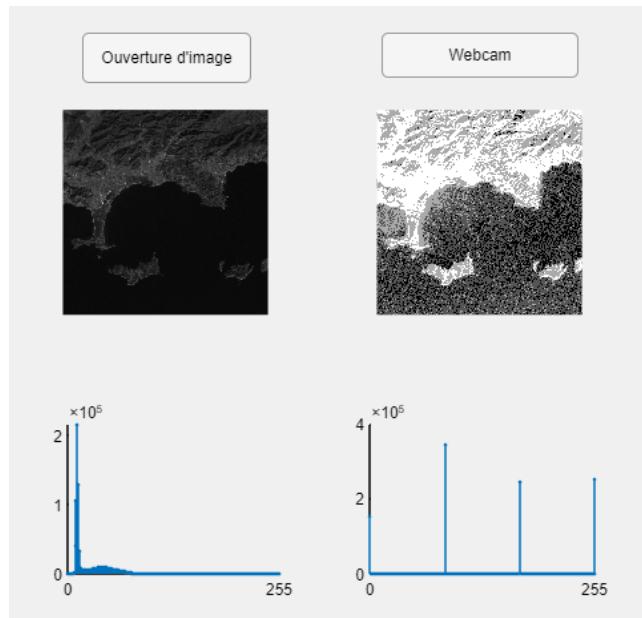


FIGURE 37 – Multiseuillage noir et blanc pour 4 niveaux

On obtient bien 4 niveaux de gris différents.

Seuillage multiseuils en couleur

Pour le multiseuillage en couleur, c'est le même principe que l'on va appliquer bande par bande à notre image. On crée une nouvelle **slider bar**, et de la même manière on obtient :

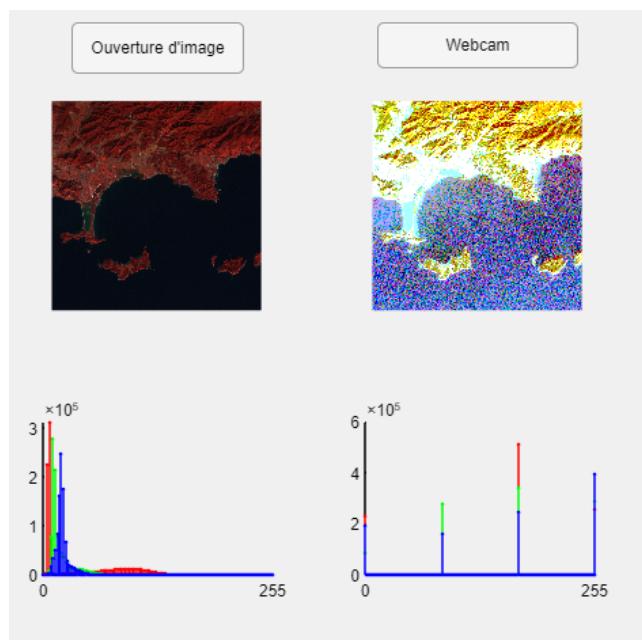


FIGURE 38 – Multiseuillage en couleur pour 4 niveaux

Les couleurs sont aussi bien réparties entre les 4 niveaux.

Morphologie mathématique

On crée des boutons pour appliquer une érosion, une dilatation, une ouverture et une fermeture.
 L'érosion consiste à prendre la valeur minimale d'une zone de pixel pour le pixel central :

```

function IM = erosion(~, img)
    img = im2gray(img);
    [nc, nl] = size(img);
    lignesSuppl = uint8(255.*ones(1, nl));
    IML = [lignesSuppl;img;lignesSuppl]; %rajout ligne supp
    [nc, nl] = size(IML);
    colSuppl = uint8(255.*ones(nc, 1)); %rajout col supp
    IMC = [colSuppl IML colSuppl];
    IM = IMC;
    [nc, nl] = size(img);

    for i = 1:nc
        for j = 1:nl
            pix = [IMC(i, j + 1), IMC(i+1, j), IMC(i+1, j+1),
                   IMC(i+1,j+2), IMC(i+2,j+1)];
            mpix = min(pix(:));
            IM(i, j) = mpix;
        end
    end
    IM(1,:,:)= [];
    IM(end,:,:)= [];
    IM(:,1,:)= [];
    IM(:,end,:)= [];
    IM = uint8(IM);
end

```

On crée un cadre de pixels supplémentaires autour de l'image afin de réaliser l'érosion aussi sur ses contours.

Ensuite, on récupère les pixels contenus dans le masque (ici de taille 3×3) puis on récupère le minimum.

On obtient comme résultat :

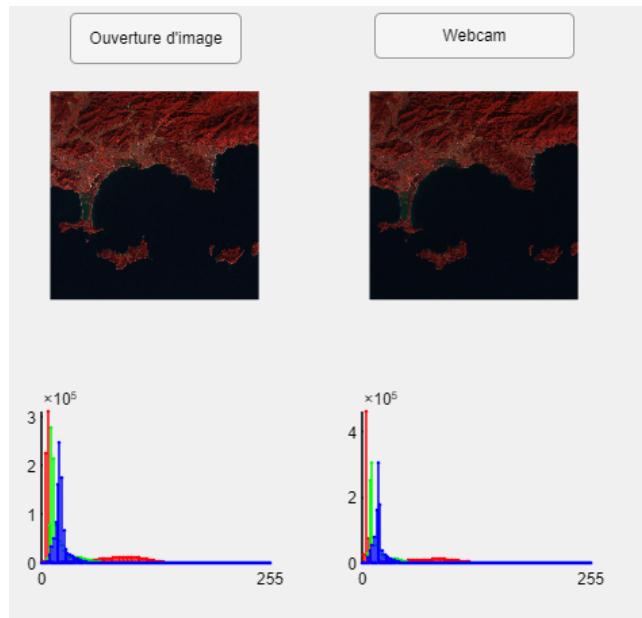


FIGURE 39 – Résultat de l'érosion

Les contrastes semblent un peu plus amplifiés, et l'image plus sombre, comme le suggère l'histogramme.

Pour la dilatation, le principe est le même sauf qu'au lieu du minimum on prendra la maximum :

```

function IM = dilatation(~, img)
    img = im2gray(img);
    [nc, nl] = size(img);
    lignesSuppl = uint8(255.*ones(1, nl));
    IML = [lignesSuppl;img;lignesSuppl]; %rajout ligne supp
    [nc, nl] = size(IML);
    colSuppl = uint8(255.*ones(nc, 1)); %rajout col supp
    IMC = [colSuppl IML colSuppl];
    IM = IMC;
    [nc, nl] = size(img);

    for i = 1:nc
        for j = 1:nl
            pix = [IMC(i, j + 1), IMC(i+1, j), IMC(i+1, j+1),
                   IMC(i+1,j+2), IMC(i+2,j+1)];
            mpix = max(pix(:));
            IM(i, j) = mpix;
        end
    end
    IM(1,:,:,:) = [];
    IM(:,end,:,:) = [];
    IM(:,1,:,:) = [];
    IM(:,:,end,:,:) = [];
    IM = uint8(IM);
end

```

On obtient comme résultat :

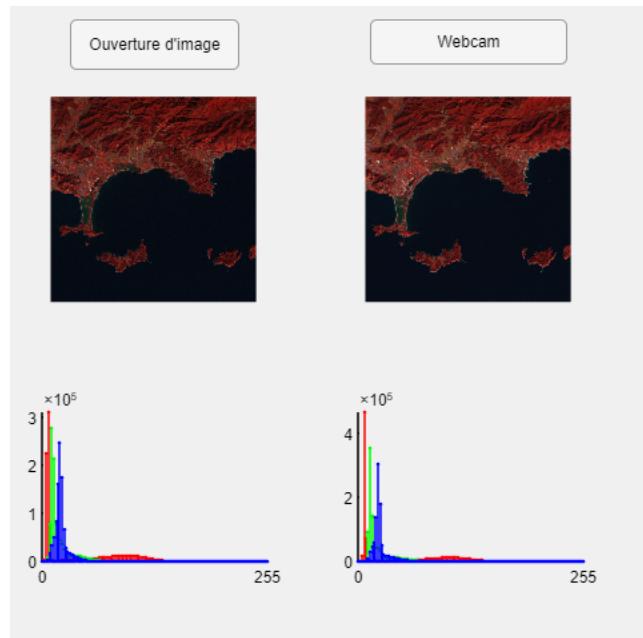


FIGURE 40 – Résultat de la dilatation

Cette fois, l'image semble plus claire, et les contrastes sont aussi amplifiés.

L'ouverture consiste à appliquer une érosion puis une dilatation à l'image érodée. On obtient le résultat suivant :

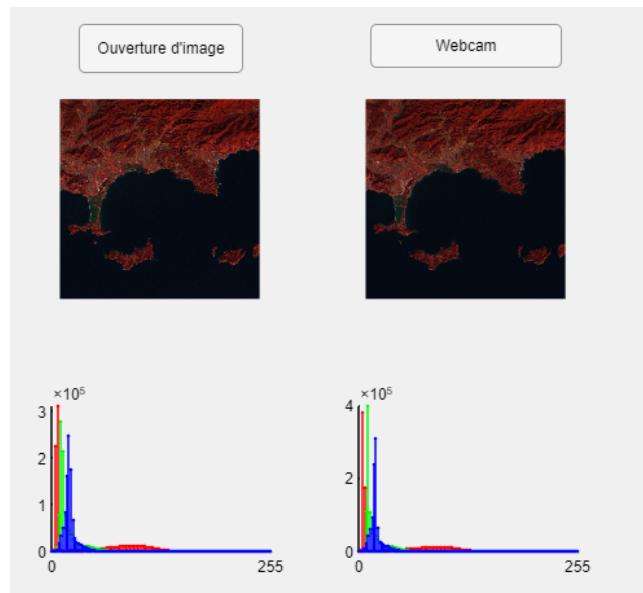


FIGURE 41 – Résultat de l'ouverture

L'image est plus sombre, et semble un peu moins nette.

Enfin, la fermeture consiste à faire l'inverse, c'est-à-dire dilater l'image puis éroder l'image dilatée.

On obtient donc :

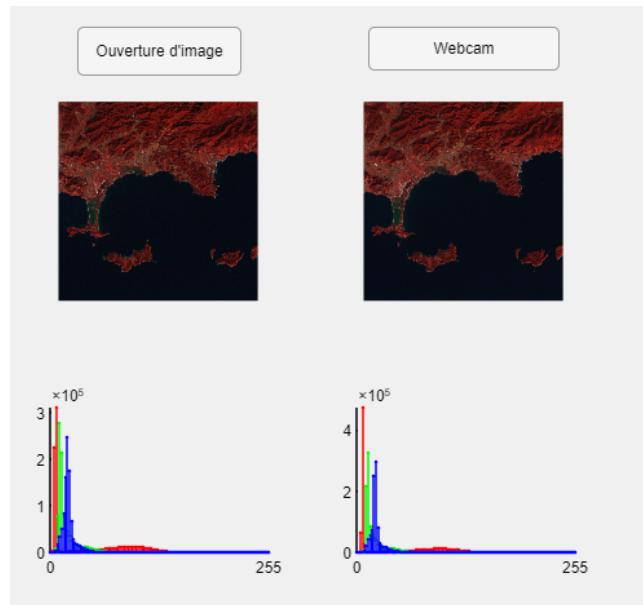


FIGURE 42 – Résultat de la fermeture

Cette fois-ci, l'image est plus claire, et aussi plus nette.

Application au filtrage de bruit

Si l'on applique un bruit poivre et sel à l'image et qu'on réalise une érosion, on obtient ceci :

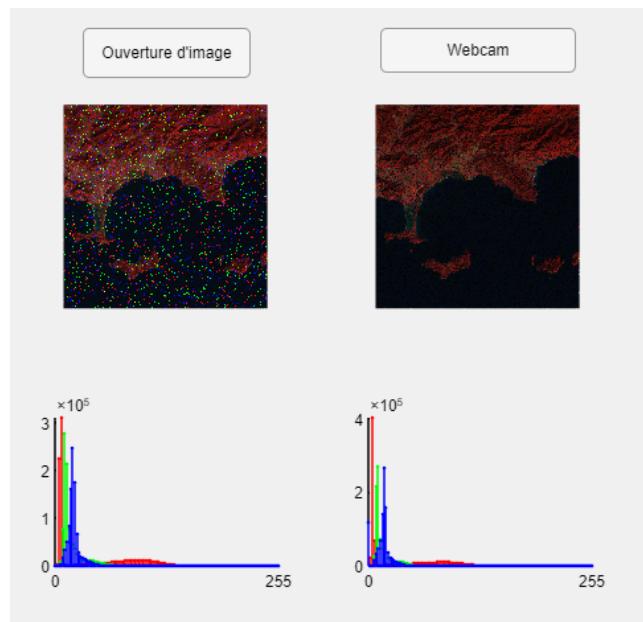


FIGURE 43 – Érosion sur un bruit poivre et sel

Le bruit en couleur semble avoir disparu, mais il a été remplacé par des pixels noirs sur le côté coloré de l'image.

Si on applique une dilatation :

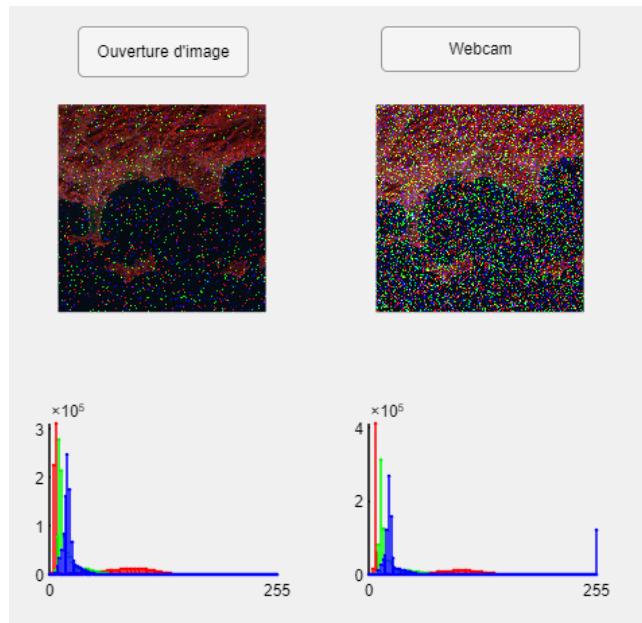


FIGURE 44 – Dilatation sur un bruit poivre et sel

Le bruit semble s'être multiplié.

Maintenant, si l'on applique une ouverture :

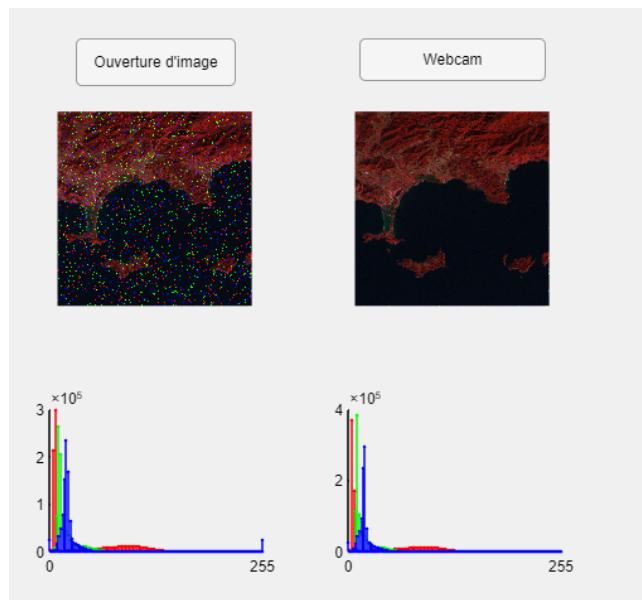


FIGURE 45 – Ouverture sur un bruit poivre et sel

Le bruit semble avoir presque totalement disparu, seuls subsistent quelques pixels noirs sur le fond coloré. On va donc créer un bouton **Morphologie mathématique** appliquant une ouverture dans notre panel "Filtrage".

Application à la détection de contours

L'idée est la même que précédemment, à l'exception que l'on applique la dilatation sur l'image originale, et que l'on calcule la différence entre image érodée et image dilatée :

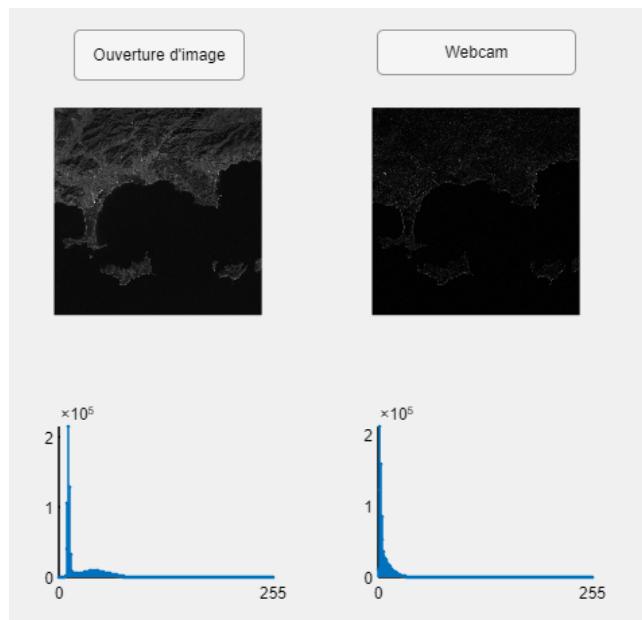


FIGURE 46 – Érosion, dilatation et différence des deux

On obtient les contours de l'image. On peut donc créer un bouton **Morphologie mathématique** dans notre panel "Contours" et y utiliser cette méthode.

Test de l'IHM sur d'autres images couleurs

Afin de tester la robustesse de notre IHM, on va prendre une image couleur aux dimensions au moins deux fois plus grandes, et y appliquer quelques fonctions :

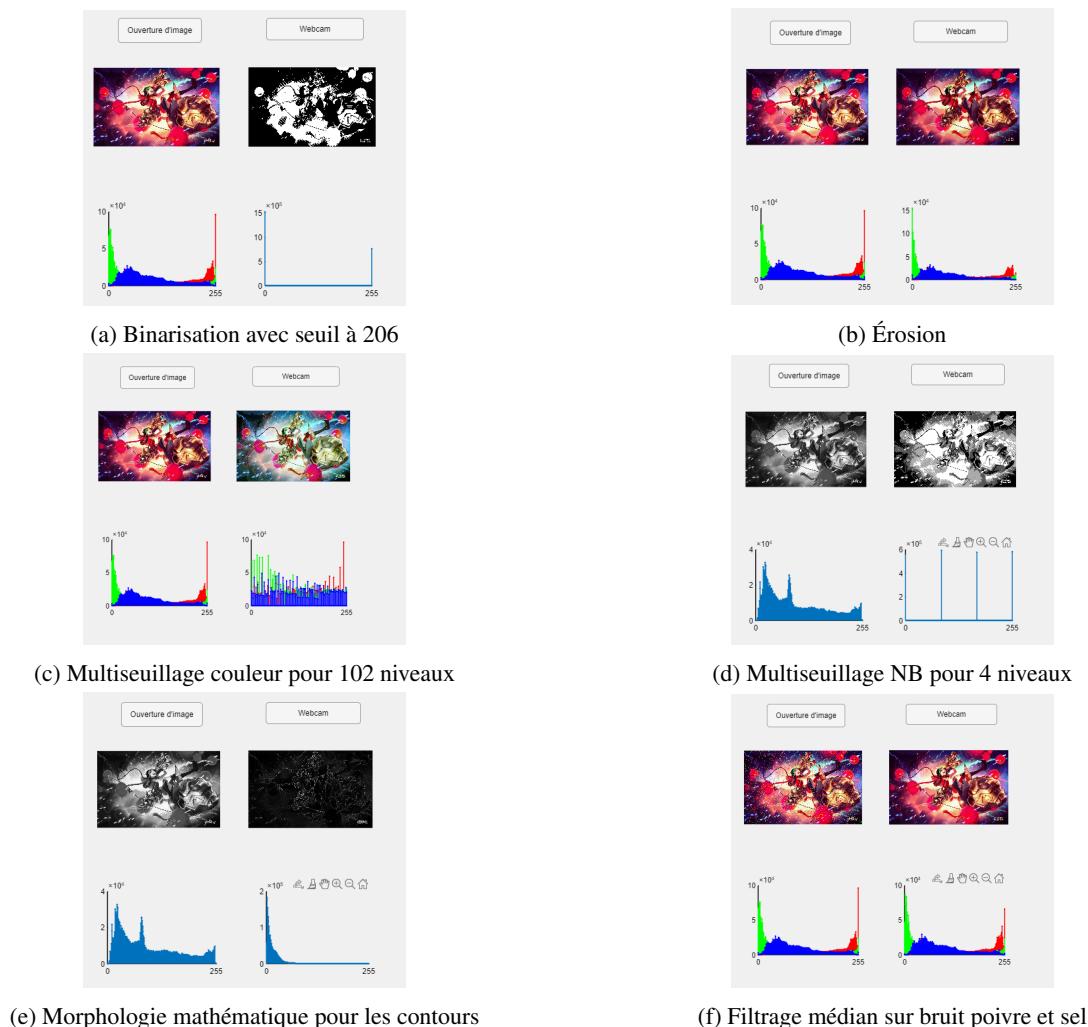


FIGURE 47 – Différents tests sur une autre image couleur

Les différents traitements fonctionnent. En cas d'image en NB, l'IHM aura malheureusement quelques dysfonctionnements.

Webcam

On peut utiliser une image prise par webcam comme entrée pour nos traitements. Pour cela, il faut avant tout connaître les paramètres de la webcam de l'ordinateur (si c'est une webcam USB, c'est un tout autre traitement) :

```
info=imaqhwinfo('winvideo')
info.DeviceInfoSupportedFormats
vid = videoinput('winvideo','HP truevision hd camera','MJPG_640x480');
preview(vid)
```

Pour un OS Windows, il faut d'abord installer le package **winvideo**, et ensuite taper la première commande. Elle va nous donner les informations concernant **winvideo**. Ensuite, la deuxième commande permet d'obtenir les différents formats d'image supportés. La troisième commande permet de créer la vidéo, en utilisant le package, le nom de la webcam (device) et le format souhaité. Enfin, on l'affiche avec la dernière ligne. Si tout fonctionne correctement, alors on peut l'intégrer à l'IHM :

```
vid = videoinput('winvideo','HP truevision hd camera','MJPG_640x480');
start(vid)
```

```
IM = getsnapshot(vid);
stop(vid)
IM = uint8(IM);
```

La fonction `getsnapshot()` permet de récupérer une image de la vidéo, dans la même idée que de prendre quelqu'un en photo. Pour du traitement vidéo en temps réel, on n'utilisera pas cette fonction (mais c'est beaucoup demander à un logiciel de ce type).

On affiche le résultat dans l'IHM, et comme précédemment, on réalise les différents traitements. Évidemment, la qualité de la webcam (du moins celle de cet ordinateur) n'est pas des plus hautes, ainsi l'image rendue n'est pas spécialement jolie et plutôt pleine de bruit. Néanmoins, les traitements vus plus hauts fonctionnent tous sans exception.

Le test de la webcam sera laissé au soin du lecteur. Avant tout, ne pas oublier de vérifier la présence du package correspondant à l'OS et le nom de la webcam de l'ordinateur !

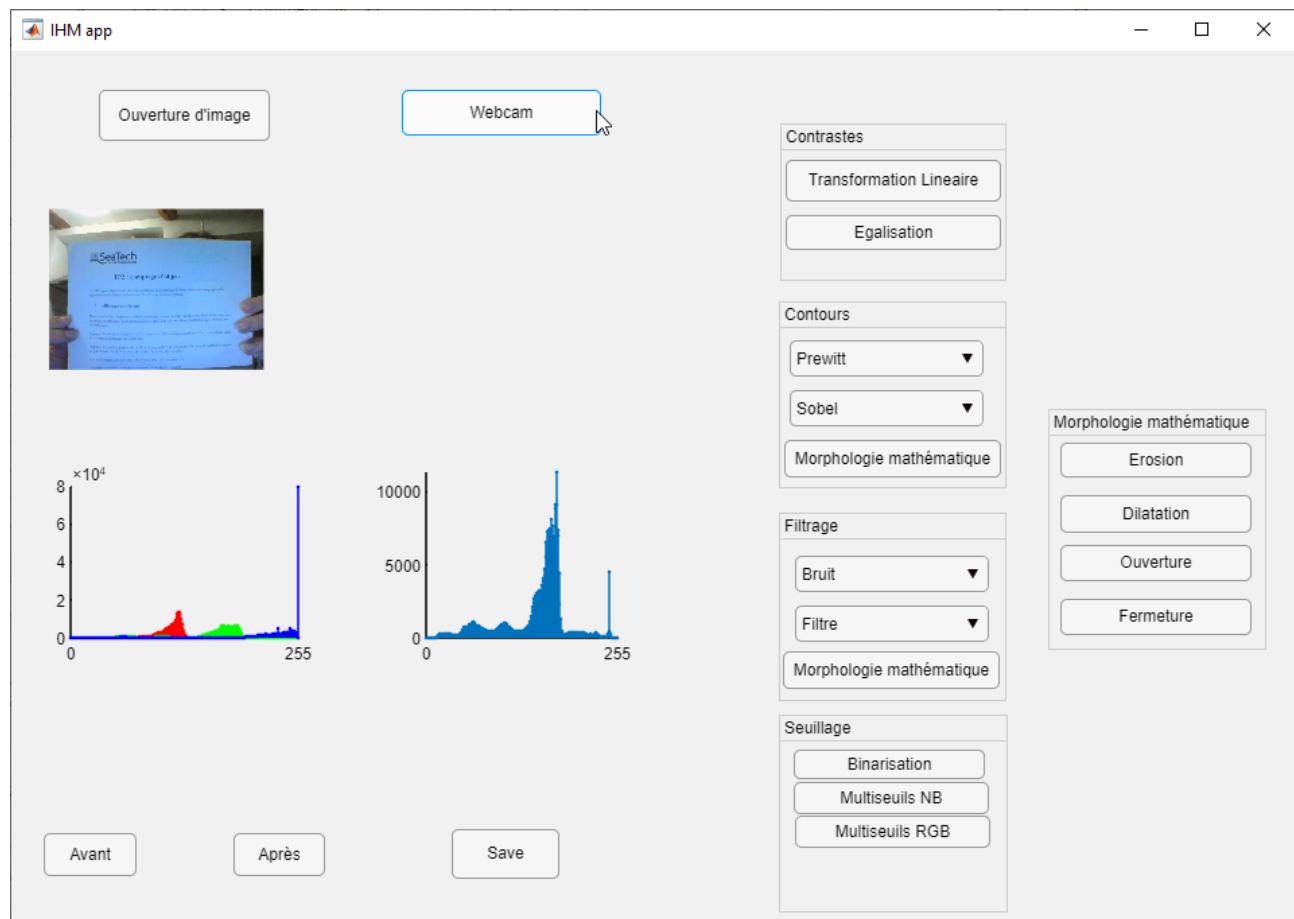


FIGURE 48 – Test de la webcam

Traitement d'image en Python

Le traitement d'image en Python s'effectue principalement avec la librairie **OpenCV**, qui permet de les traiter et les afficher.

Pour tout ce qui est affichage de graphes et tracé de courbes, on utilise la librairie **Matplotlib**, qui est la version Python de MatLab.

Enfin, pour des opérations vectorielles, matricielles, et les tableaux, on utilise la librairie **Numpy**.

Afin d'exécuter un programme Python, on lance une invite de commande et on tape la commande :

```
python nomduprogramme.py
```

Une fois notre fichier .py créé, on importe les librairies citées ci-dessus :

```
import numpy
import cv2
import matplotlib.pyplot as plt
```

De cette manière, on pourra utiliser les fonctions nécessaires en appelant les librairies par leur nom ou leur nom choisi avec **as**.

cv2 est le module de la librairie OpenCV.

On souhaite afficher une image. Pour cela, on utilise la fonction *imread()* de la librairie OpenCV :

```
img = cv2.imread("test_5bb_blancs.png")
cv2.imshow("Image originale", img)
cv2.waitKey(0)
```

Soit notre image chargée dans **img** avec la fonction *imread()*. La fonction *imshow()* crée une fenêtre du nom du premier paramètre, et y affiche l'image en deuxième paramètre.

Enfin, la dernière ligne permet de laisser la fenêtre ouverte tant qu'on ne clique pas sur "fermer la fenêtre" (la croix) :

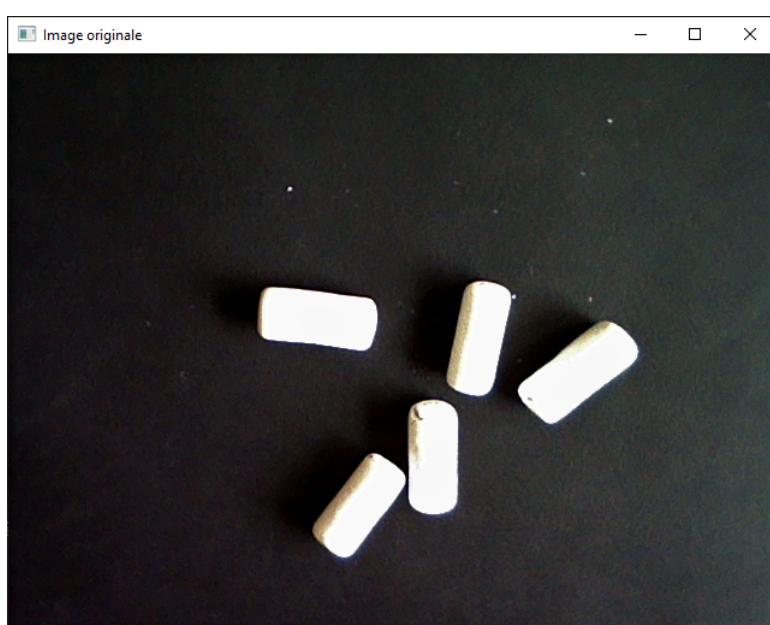


FIGURE 49 – Fenêtre affichant l'image lue avec *imread()*

On souhaite maintenant la rendre en noir et blanc. On crée une fonction dans le même fichier, `rgb_to_bw()` :

```
def rgb_to_bw(img):
    nl = img.shape[0]
    nc = img.shape[1]
    IM = img[:, :, 0]
    for i in range(nl):
        for j in range(nc):
            IM[i, j] = (int(img[i, j, 0]) + int(img[i, j, 1]) + int(img[i, j, 2])) / 3

    return IM
```

On récupère les dimensions de l'image en paramètre (celle que l'on veut rendre en NB) avec la fonction `shape()`. Cette fonction récupère et stocke dans un vecteur à 3 dimensions, dans l'ordre, le nombre de lignes, de colonnes et de bandes de l'image. Si l'on souhaite récupérer seulement une valeur du vecteur, on utilise la syntaxe **[indice]**, qui nous retournera la valeur à l'indice choisi.

Une fois les dimensions de l'image source récupérée, on crée une première image de sortie **IM**, avec les mêmes dimensions, et une seule bande. En effet, une image en NB n'en possède qu'une contrairement à une couleur qui en a 3.

Enfin, on crée une double boucle `for` allant de 0 au nombre de lignes puis de 0 au nombre de colonnes, et on affecte à l'image de sortie la moyenne des pixels de chaque bande.
Pour appliquer notre fonction et observer le résultat :

```
IM = rgb_to_bw(img)

cv2.imshow("En noir et blanc", IM)
cv2.waitKey(0)
```

Soit **IM** notre image en niveaux de gris, et **img** l'image couleur. Lorsque l'on affiche **IM**, on obtient :

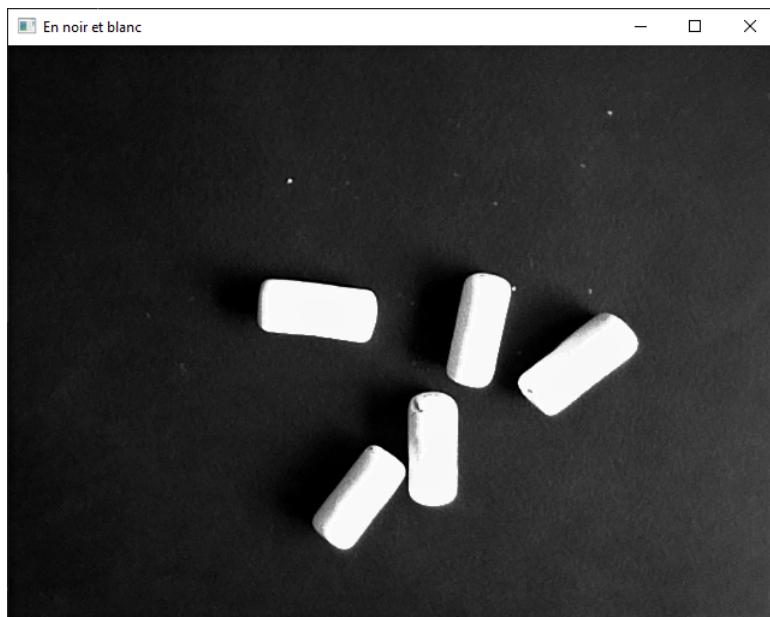
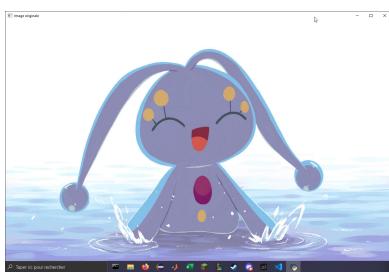


FIGURE 50 – Résultat de la conversion couleur-NB

Le résultat n'est pas flagrant car l'image originale est déjà assez binaire. On peut essayer avec une

autre image :



(a) Image couleur



(b) Image NB

FIGURE 51 – Test sur une image plus en couleur

La fonction de conversion semble efficace.

On souhaite maintenant calculer l'histogramme de cette image en niveaux de gris :

```
def histogramme(img):
    nl = img.shape[0]
    nc = img.shape[1]
    h = numpy.zeros(256)
    for i in range(nl):
        for j in range(nc):
            val = img[i, j]
            h[val] = h[val] + 1
    return h
```

Comme précédemment, on récupère les dimensions de l'image, puis on crée un vecteur de taille 256 avec la fonction `zeros()`, qui contiendra notre histogramme.

Dans une double boucle `for`, on récupère chaque pixel de l'image, que l'on place dans **h**. Chaque fois qu'une nouvelle valeur est trouvée, elle est ajoutée dans l'histogramme, soit en tant que nouvelle, soit en incrémentation d'une ancienne.

Pour l'afficher, comme c'est un graphe, on utilise la librairie **Matplotlib** :

```
h = histogramme(IM)
plp.stem(h)
plp.show()
plp.close("all")
```

Une fois l'histogramme calculé, on l'affiche avec la fonction `stem()`, soit non en courbe mais en "bâtons", puis on crée la fenêtre d'affichage avec `show()`.

La dernière ligne permet de fermer les fenêtres.

On obtient, pour l'image en NB :

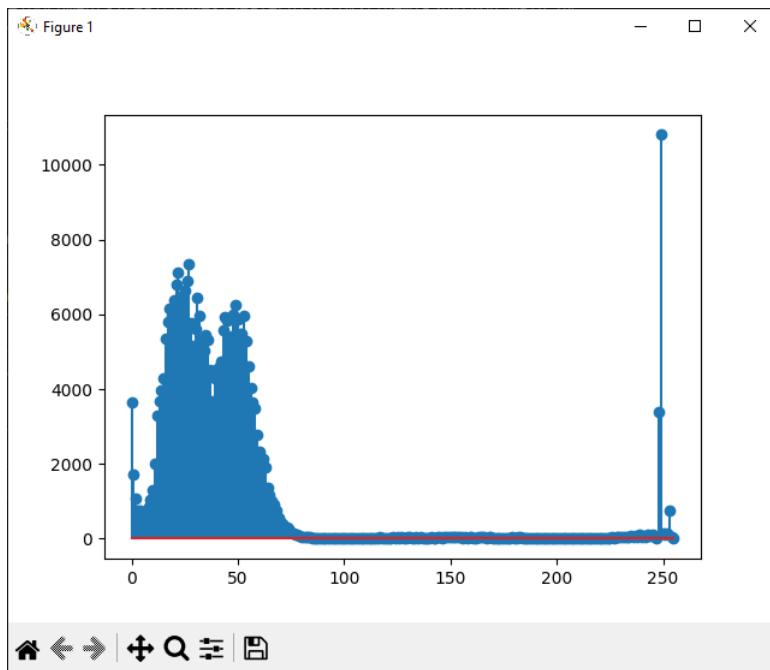


FIGURE 52 – Histogramme de l'image en NB

D'après l'histogramme, la majorité des pixels ont une valeur basse, de 0 à 100, puis une petite partie est dans le blanc.

Cet histogramme bi-modal nous permet de déterminer un seuil, afin de réaliser une binarisation :

```
def binarisation(img, seuil):
    nl = img.shape[0]
    nc = img.shape[1]
    for i in range(nl):
        for j in range(nc):
            if (img[i, j] > seuil):
                IM[i, j] = 255
            else:
                IM[i, j] = 0
    return IM
```

L'idée est répartir tous les pixels de l'image dans les valeurs 0 et 255, soit en noir et blanc. Pour cela, dans une double boucle *for*, si les pixels ont une valeur au-dessus du seuil, ils seront mis à 255, sinon à 0.

Pour l'image traitée précédemment, selon son histogramme, on choisit un seuil à 100, puis on affiche l'image comme plus haut :

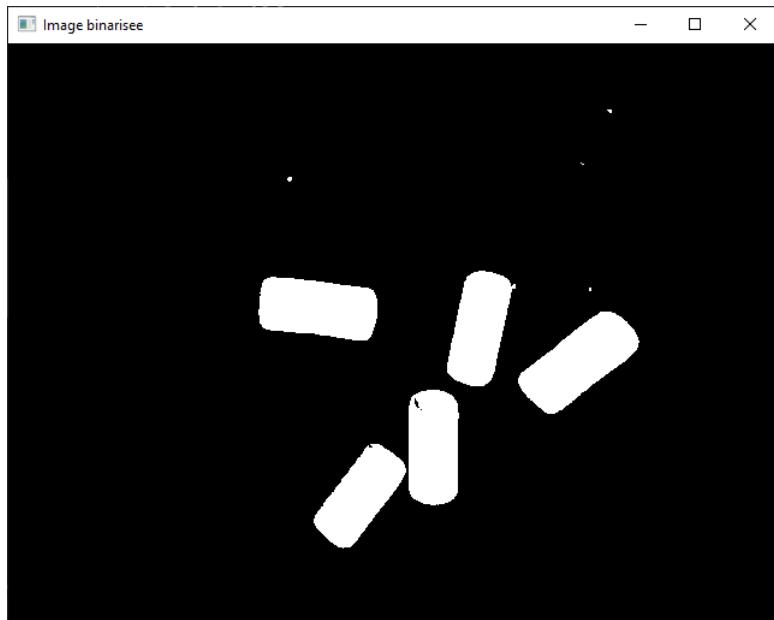


FIGURE 53 – Image binarisée

L'image ne comporte plus que 2 valeurs différentes : 0 et 255. Ce traitement nous permet de séparer les bonbons du reste, même s'il reste des pixels blancs "parasites" disséminés sur l'image, qui ne sont évidemment pas des bonbons.

Lorsque l'on calcule l'histogramme de l'image, on observe qu'elle vérifie bien la binarisation :

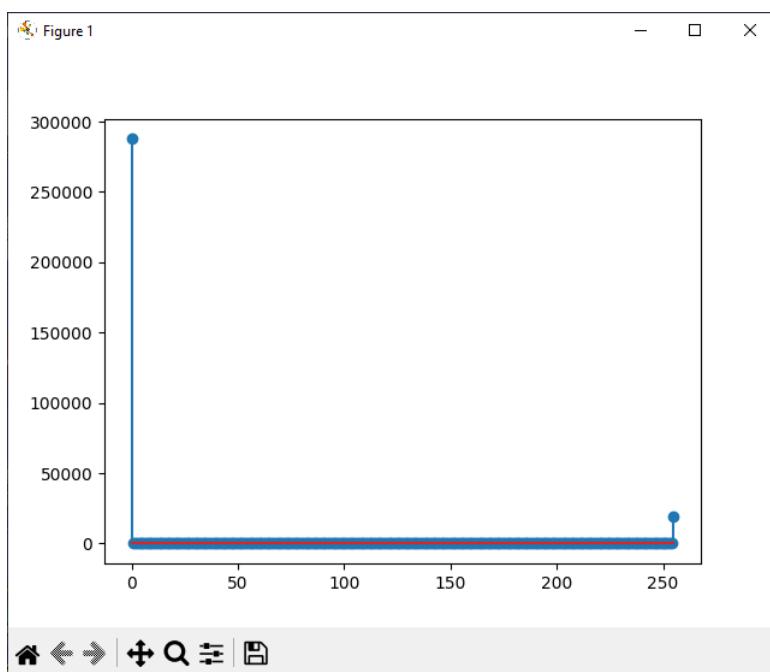


FIGURE 54 – Histogramme de l'image binarisée

Cet histogramme permet de déterminer le nombre exact de pixels blancs en récupérant $h[255]$, puis ensuite de les diviser par le nombre de bonbons. On obtiendra de cette manière le nombre de pixels blancs par bonbon :

```
nombre_pixels_blancs = h_bin[255]/5
```

```
nombre_bonbons = h_bin[255]/(nombre_pixels_blancs)
print(nombre_bonbons)
```

Le nombre de pixels blancs par bonbon déterminé est celui pour l'image avec 5 bonbons. Lorsque l'on affiche le résultat, on obtient évidemment 5 bonbons.

Si on garde le résultat de *nombre_pixels_blancs* pour d'autres images de bonbons, on n'aura pas forcément le bon résultat. La principale raison est que les bonbons sur cette image sont pris selon un certain angle, laissant une certaine surface blanche. Si les bonbons sont pris selon un autre angle, il y aura plus ou moins de pixels blancs par bonbon.

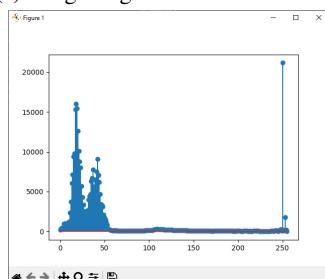
Par exemple, pour une image avec 11 bonbons :



(a) Image originale avec 11 bonbons



(b) 11 bonbons en NB



(c) Histogramme des 11 bonbons en NB



(d) 11 bonbons binarisés, seuil à 100

FIGURE 55 – Image avec 11 bonbons

Il y a, d'après le calcul, 9.9 bonbons, pour 11 réels. Le seuil choisi à 100 d'après l'histogramme peut être descendu à 80, et on obtiendra 10.4 bonbons.

Il sera difficile d'obtenir le nombre réel de bonbons avec cette méthode.

On peut réaliser le même traitement pour une image avec un seul bonbon :

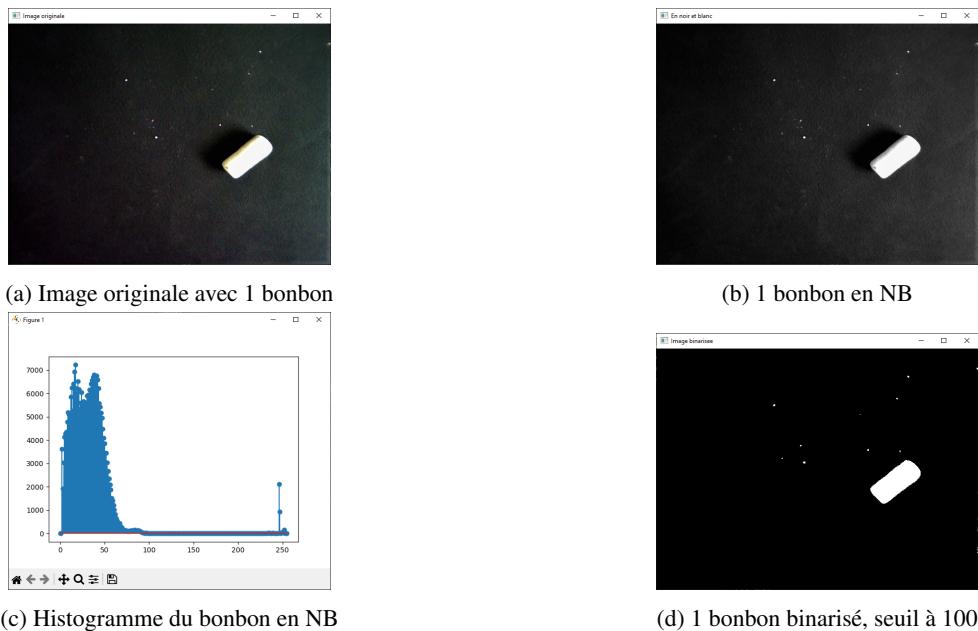


FIGURE 56 – Image avec 1 bonbon

Pour un seuil à 100, on obtient 1.14 bonbons. Le résultat n'est pas trop mauvais, mais est bien meilleur pour un seuil à 150 où l'on obtient 1.03 bonbons. On pourrait réaliser une méthode qui calculerait une moyenne des pixels blancs par bonbons pour en déterminer le nombre sur l'image, avec des critères de nombre de pixels minimum/maximum pour compter un bonbon.

Conclusion

Le traitement d'image en Matlab est plus adapté d'un point de vue traitement de matrice, même s'il est tout à fait possible de le réaliser en Python.

En ce qui concerne l'IHM, le créer en Python aurait demandé l'utilisation de librairies supplémentaires comme **tkinter** qui permet de créer une interface à partir de rien. Si l'on veut y passer moins de temps ou que l'on débute, le plus simple est d'utiliser l'app designer de Matlab qui est un outil assez complet.